THE UNIFRAME SYSTEM-LEVEL

GENERATIVE PROGRAMMING FRAMEWORK

A Thesis

Submitted to the Faculty

of

Purdue University

by

Zhisheng Huang

In Partial Fulfillment of the

Requirements for the Degree

of

Master of Science

August 2003

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE
AUG 2003

2. REPORT TYPE

3. DATES COVERED
00-00-2003 to 00-00-2003

4. TITLE AND SUBTITLE
The Uniframe System-Level Generative Programming Framework

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Indiana University/Purdue University,Department of Computer and Information Sciences,Indianapolis,IN,46202

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT
see report

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			
unclassified	**unclassified**	**unclassified**	**Same as Report (SAR)**	**278**	

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std Z39-18

To Ping.

ACKNOWLEDGEMENTS

The graduate education in the MS program at the Department of Computer and Information Science of Indiana University-Purdue University Indianapolis (IUPUI) has been a turning point and great experience in my life. It helps me to realize my potential as a good computer professional. The knowledge gained will be valuable to me throughout my career. My work as a Research Assistant on the UniFrame research project was a great opportunity to further enhance the skills acquired during my graduate study. I would like to take this opportunity to express my sincere gratitude to all those who helped to make my graduate study fruitful and make this thesis possible.

I would like to profoundly thank my advisors Dr. Rajeev Raje and Dr. Andrew Olson for their great guidance throughout the course of my study and research work. Their immense inputs and insights were invaluable for this thesis. I am very grateful to them for their constant encouragement, making me reach higher in all my academic endeavors.

I would also like to thank Dr. Jeffery Huang for being on my thesis committee and for the effort to review my thesis. I am also very grateful to him for his encouragement in my endeavor in the computer science.

I would like to thank all my colleagues on the UniFrame project, the faculty and staff of the Department of Computer and Information Science of IUPUI for their assistance towards this thesis.

I am thankful to the U.S. Department of Defense and the U.S. Office of Naval Research for supporting this research under award number N00014-01-1-0746.

Finally, I would like to thank my wife for her great love, encouragement and support, which have been the source of my inspiration and strength all the time.

TABLE OF CONTENTS

Page

Page

Page

LIST OF TABLES

Table Page

Table Page

LIST OF FIGURES

ABSTRACT

Huang, Zhisheng. M.S., Purdue University, May 2003. The UniFrame System-Level Generative Programming Framework. Major Professors: Dr. Rajeev Raje and Dr. Andrew Olson.

Current and future distributed computing systems (DCS) will certainly require combining heterogeneous software components that are geographically dispersed so that their realizations not only meet the functional requirements, but also satisfy the non-functional criteria such as the desired quality of service (QoS). The UniFrame Approach (UA) incorporates the concepts of a meta-component model, generative programming and QoS, to achieve a semi-automatic software development for DCS. It permits a large degree of component reuse and a seamless interoperation while creating QoS-aware DCS. UA has two levels, the component level and the system level. This thesis presents the UniFrame System-Level Generative Programming Framework (USGPF). The proposed USGPF addresses the following issues: 1) a promising shift in the paradigm of developing DCS from single systems to families of systems; and 2) a framework at the system level for developing QoS-aware DCS. The USGPF consists of three parts: 1) the UniFrame Generative Domain Model (UGDM), which captures the common and variable properties of a DCS family; 2) the UniFrame UGDM Development Process (UGDP), which is a use-case driven, architecture-centric, iterative and incremental process to create a UGDM for a DCS family; and 3) the UniFrame System Generation Infrastructure (USGI), which has a built-in support for the QoS validation to assist in the creation of QoS-aware DCS. A prototype is designed and implemented to validate the proposed USGPF. The results of applying this approach in the semi-automatic construction of simple DCS from a banking domain are promising and demonstrate the effectiveness of this research.

1. INTRODUCTION

The software development has been steadily evolving during the past few decades. There has been a constant endeavor to bring the software industry on par with its more mature peers like the hardware industry. The emergence of the component-based software development (CBSD) and product line practice (PLP) are concrete steps in this direction.

For many years, software systems were built individually for specific purposes. With the advent of Object-Oriented Programming the concept of code reuse became a highly popular and cost-effective programming technique. The CBSD takes this step further by developing the entire software systems from appropriate commercial-off-the-shelf (COTS) software components. Szyperski [SZY99] defines a software component as a unit of composition with contractually specified interfaces and explicit context dependencies. At the same time, with the advent of high speed networks and the growing popularity and availability of the Internet, the paradigm in software development is shifting towards distributed computing. CBSD has been a growing trend in the development of software solutions for distributed computing systems (DCS). In recent years, the software development has also shifted from the development of a single system to the development of a family of systems. Generative programming [CZA00] is the technique for developing such system families. The product line practice (PLP) initiative [SEI02] launched by the Software Engineering Institute (SEI), Carnegie Mellon University, is an attempt to facilitate this transition. The quick advances in the software development not only open a lot of opportunities but also pose enormous challenges, especially for the development of DCS. This thesis tries to address some of these challenges.

1.1 Problem Definition and Motivation

As distributed computing becomes more and more crucial for the success of today's enterprises, there is an increasing need to develop software for DCS in an effective and efficient way.

However, many challenges arise during the application of the CBSD to DCS. Some of these challenges are an effect of the presence of multiple component models. Currently, different component models have been proposed, such as Java™ Remote Method Invocation (RMI) [ORF98], Common Object Request Broker Architecture (CORBA™) [OMG99, ORF98, SEI96], Distributed Component Object Model (DCOM™) [MS98], and .NET [NET03]. There are difficulties in bridging the components belonging to different models, thus reducing the degree of component reuse. How to seamlessly and effectively create DCS from heterogeneous distributed software components based on these different models is a challenge that is currently being addressed by the research community.

Another challenging issue is regarding the quality of service (QoS) of components or of DCS generated from components. The ISO defines QoS as the totality of features and characteristics of a product or a service that bear on its ability to satisfy stated or implied needs [ISO86]. In order for a development approach to generate DCS with predictable quality, the approach should have a built-in support for the QoS. However, currently there are no widely accepted frameworks that incorporate QoS as an inherent part of DCS development. This can lead to inconsistencies and irregularities in the quality of DCS. This calls for a concrete framework which incorporates the QoS as an inherent part of DCS development process and offers objective means to quantify, verify, validate and specify the QoS of DCS.

The use of components to develop software for DCS is consistent with the notions of generative programming and the product line practice (PLP). However, despite the advances in the software development and the notion of generative programming, a lot of distributed computing systems are still designed and built as single systems. This paradigm of single system development has the problems of large investment, long development cycles, difficulties in the system integration, and a lack of predictable quality [COH00]. One reason of the delay in the application of the generative programming to the distributed computing is due to the inherent complexity of DCS. Another reason is that there is no well-defined process for creating DCS in such a way to meet the increasing demand of more reliable DCS. The existing development processes lack the built-in QoS support that is necessary for creating QoS-aware DCS. Hence, it is utmost necessary to propose a development process that will incorporate these features.

The recent shift in the focus of Object Management Group (OMG) to the Model Driven Architecture (MDA) [OMG01] is a recognition that the bridging of heterogeneous software components based on different component models requires the standardization not only of the infrastructure but also of the business and component meta-models. With MDA, the development of DCS focuses first on the functionality and behavior, undistorted by idiosyncrasies of the technology or technologies in which it will be implemented. Thus, MDA divorces implementation details from the business functions. So, it is not necessary to repeat the process of modeling an application or system's functionality and its behavior each time a new technology is created. With MDA, the functionality and the behavior are modeled once, and the mapping from a platform independent model (PIM) to a platform dependent model (PSM) is implemented by tools, easing the task of supporting new technologies.

Web Services [WEB02] are viewed as another possible solution to the problem of bridging diverse heterogeneous distributed component models. Mayo [MAY02] describes Web Services as a standards-based software technology that lets programmers and integrators combine existing and new systems or applications in new ways over the Internet, within a company's boundaries, or across many companies. Web Services allow interoperability between the software written in different programming languages, developed by different vendors, or running on different Operating Systems or platforms. Thus, Web Services provides the flexibility with respect to the interoperability, reuse and development of applications in a distributed environment.

However, both MDA and Web Services do not take into account the QoS of components and/or systems. They also do not define the process with a built-in QoS support to create a DCS family. The Unified Meta-Component Model Framework

(UniFrame) research [RAJ00, RAJ01, RAJ02] is another attempt which aims to address all above listed challenges. The UniFrame Approach (UA), a key constituent of the UniFrame, tries to unify the existing and emerging distributed component models under a common meta-model, the Unified Meta-component Model (UMM). It has the following key concepts: a) a meta-component model (the Unified Meta Model – UMM [RAJ00]), with an associated hierarchical setup for indicating the contracts and constraints of the components, b) an integration of the QoS at the individual component and distributed application levels, c) the validation and assurance of the QoS, based on the concept of event grammars, and e) generative rules with formal specifications to assemble a DCS from an ensemble of components out of available component choices. Chapter 3 provides a more detailed overview of this approach.

The application of UA to create DCS has two levels [RAJ01]: 1) component level - in this level, different components are created by developers, tested and verified from the point of view of QoS, and then deployed on the network; 2) system level - this level concentrates on creating a generative domain model (GDM) and automatically or semi-automatically generating DCS by assembling a collection of heterogeneous distributed software components based on the GDM. The generative programming techniques can be applied at both levels of the UA. This thesis focuses on applying the generative programming techniques at the system level in the context of UA.

1.2 Objectives

Specifically, this thesis aims at proposing the UniFrame System-Level Generative Programming Framework (USGPF) to address the challenges stated in the previous section. The overall objectives of the USGPF are:

- To propose the UniFrame Generative Domain Model (UGDM) to capture the common and variable properties of a DCS family with QoS concerns in the solution space. The generative domain model (GDM) for a DCS family differs significantly from a model for a standalone program. The UGDM should be able to capture many aspects of DCS in order to assist the developing of more reliable

DCS or QoS-aware DCS. The UGDM should take into account various aspects of DCS, like system architecture, component interactions, communication patterns, QoS composition and decomposition, and event grammar.

- To create the UniFrame UGDM Development Process (UGDP) for the development of a DCS family by incorporating generative programming techniques into the UA at the system level. The UGDP should be an effective process for developing UGDM for any target domain. It should have a built-in support to incorporate the QoS into UGDM in order to develop quality-oriented and time-to-market DCS with lower development and maintenance costs.
- To create the UniFrame System Generation Infrastructure (USGI) to assist in the generation of QoS-aware DCS during the phase of application engineering based on the UGDM. The proposed USGI should have a flexible architecture and should be platform independent. The USGI replaces the manual search for, and adaptation and assembly of, heterogeneous and distributed components with automation. It should support the generation of DCS automatically to the extent feasible and should have the built-in support for the system QoS validation.

1.3 Contributions

The contributions of this thesis are:

- Definition of the UniFrame Generative Domain Model (UGDM). The UGDM has an inherent consideration of the QoS requirements to assist the need of developing QoS-aware DCS. The proposed UGDM consists of a set of models to represent different aspects of a DCS family to assist the automatic system generation and QoS validation.
- Definition of the UniFrame Domain Specific Language (UDSL) to document various models in the UGDM in an informal fashion.
- Creation of the UniFrame UGDM Development Process (UGDP) to formulate a UGDM in assisting the development of a DCS family. The UGDP is a use-case

driven, architecture-centric and iterative process. It has a built-in support to integrate QoS into the UGDM.

- Development of a platform independent UniFrame System Generation Infrastructure (USGI) for efficiently generating QoS-aware DCS by seamlessly integrating heterogeneous distributed software components.
- Validation of the above mentioned objectives by a detailed case study involving an example from the banking domain.

1.4 Thesis Organization

This thesis is organized into eight chapters. Chapter 1 provides an introduction with the problem definition and motivation, objectives, contributions and thesis outline. Chapter 2 presents the related work on the generative programming, domain engineering and application engineering. Chapter 3 provides an overview of the UniFrame research project, which is the context for this thesis. This chapter also outlines the UniFrame System-Level Generative Programming Framework (USGPF). Chapter 4-6 describes the USGPF in detail. USGPF consists of three parts: UniFrame Generative Domain Model (UGDM), UniFrame UGDM Development Process (UGDP) and UniFrame System Generation Infrastructure (USGI). Chapter 4-6 describes these three parts respectively. Chapter 7 describes the design and implementation of a prototype for the USGI. An example from the banking domain, which serves as the case study for the USGPF, is developed and demonstrated throughout Chapter 4, Chapter 5 and Chapter 7. Chapter 8 provides a discussion of the features of the USGPF, possible enhancement for the USGPF as future work and a summary of this thesis.

2. BACKGROUND AND RELATED WORK

In the previous chapter a brief introduction is presented, along with the problem definition, objectives and contributions of this thesis. This chapter provides an overview of the background and the related work that has influenced the development of the USGPF.

2.1 Generative Programming

The generative programming is concerned with bringing the automation to the software development. The goal of the generative programming is to be able to automatically generate systems from a system family based on given specifications. A system family is a group of systems that can be built from a common set of assets. The achievement of this goal requires the development of a model of the system family, a way to specify system requirements, the availability of components from which the system can be assembled, and means of mapping the problem specification onto the required components (out of the available ones) to generate the system using a configuration generator (or system generator).

In [CZA00], the generative programming paradigm is formally defined as: "Generative Programming is about manufacturing software products out of components in an automated way. It requires two steps: a) a design and implementation of a generative domain model, representing a family of software systems (development for reuse). This model includes also a domain-specific software generator; b) given a particular requirements specification, a highly customized and optimized end-product can be automatically manufactured from implementation components by means of generation rules (development with reuse)". The methods presented in [CZA00] can be applied both "in the small", i.e., at the level of classes and procedures and "in the large", to develop families of large systems.

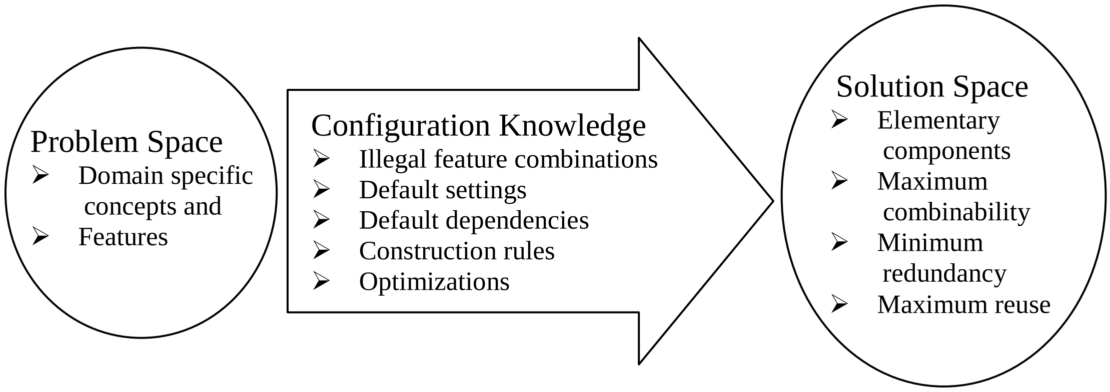


Figure 2.1 Elements of a Generative Domain Model
(from [CZA00, CZA99])

The generative programming requires the development of a generative domain model (GDM). This model consists of a problem space, a solution space, and the necessary configuration knowledge to map them together (see Figure 2.1). The problem space consists of application concepts and features that an application programmer can use to express the requirements for generating systems from a system family. This problem space can be explored using techniques from the domain engineering. The solution space is made up of the component implementations in all of their potential combinations. The configuration knowledge takes into account considerations such as illegal feature combinations, default settings, default dependencies, construction rules, and optimization rules. Configuration generators (or system generators, often referred to simply as generators) are created to implement this knowledge. A configuration generator is responsible for checking to see if the system can be built, completing the specification by computing defaults, and assembling the implementation components. An important concept to keep in mind when designing the problem space is that application programmers should only be required to specify as much information as is necessary to identify potentially appropriate components from the generative library. The programmers should be allowed to specify details or elect to supply some of his own implementations for specific functionalities if desired.

An important advantage of the separation between the problem space and solution space is the possibility to evolve both spaces in a relatively independent way. In particular, new components can be added to the solution space or the existing ones can be improved. As long as the new components or the improved components can cover the functionality delineated by the problem space, the existing client code can remain unaltered. This is so because the client code orders systems and components by means of the language of the problem space, and the generator takes care of the mapping of the problem specifications onto the configurations of the new components. Thus, adding new components only requires modifying the generator. However, this task may not be trivial. In the UniFrame Approach, which will be reviewed in detail in next chapter, with the service of the active component management, which dynamically and actively discovers and registers components deployed over the network, components are separated from the generator. Thus, when new components are deployed on the network, no modification is needed for the system generator.

The main steps necessary in the generative programming are identified in [CZA00]:

- Domain scoping
- Feature and concept modeling
- Designing a common architecture and identifying implementation concepts
- Specifying domain specific notations for ordering systems
- Specifying the configuration knowledge
- Implementing the components
- Implementing the domain specific notations
- Implementing the configuration knowledge using generators

These steps specify what needs to be done when applying the generative programming, but not in what order. It is best to perform these steps iteratively and incrementally. These steps are reflected in the USGPF and the meanings of each step will be discussed in the context of USGPF from Chapter 4 to Chapter 7.

2.2 Product Line Practice

In the component-based software development (CBSD), the domain engineering phase and the component engineering phase covers the development of reusable assets (including system architecture, component code, etc) and a production plan for producing concrete systems from these assets. In the phase of application engineering concrete systems are generated from these assets. However, in order to successfully introduce and run CBSD in an organization, a lot of issues have to be addressed. In particular, there are management and organizational issues concerning the process and the feedback between different phases. There are concerns about how to successfully transit to a system-family-oriented development, how to launch and institutionalize it and how to manage the associated risks. In addition, in order to determine what features are needed now and in the future, issues like the market analysis and the technology forecasting also need to be addressed. Furthermore, supports are needed to decide whether to develop components in-house or to purchase Commercial Off-The-Shelf (COTS) and Commercial Off-The-Net (COTN) components. Methods to evaluate and test architectures, components, generic and generative models, are also needed. These issues go beyond the scope of current component-based software engineering methods [CZA00].

The Product Line Practice (PLP) is directly connected with the generative programming technique. In 1997, the PLP initiative [SEI02] was launched by the Software Engineering Institute (SEI), Carnegie Mellon University, to address the different issues discussed in the previous paragraph. The intention was to help facilitate and accelerate the transition from the traditional single system development to sound software engineering practices using a product line approach. In PLP, a software product line is defined to be a set of software-intensive systems sharing a common, managed set of features that satisfy specific needs of a selected market or mission, and that are developed from a common set of core assets in a prescribed way [CLE01, COH00]. A software product line has the same meaning as a system family in the generative programming. The SEI's PLP Framework is the first formal attempt to codify the comprehensive information about successful product lines.

The idea behind the PLP framework is to identify the different issues and practices relevant to establishing and running successful product lines in an organization. More information can be found on the PLP Framework website [SEI02a]. The framework is documented in a living guidebook, which addresses the different practice areas and contains references to various approaches, methods, case studies, and other materials. The guidebook is being constantly updated based on a series of workshops run by SEI. It is available at www.sei.cmu.edu/plp/.

2.3 Domain Engineering Methods and Technologies

The previous two sections briefly described the generative programming technique for creating a system family and the PLP framework for helping the transition in this direction. This section provides a brief overview about prominent domain engineering methods and technologies that has influenced the development of USGPF. Typically, proposals for large scale software reuse usually introduce a concept of a software component, along with a design and implementational framework, which allows for component compositions. All the methods and technologies discussed in this section reflect this concept and use generative programming techniques. However, none of them specifically targets DCS and none of them address the heterogeneity in the distributed computing environment. Furthermore, none of them addresses the QoS issue. The effective solution to address these issues in the USGPF distinguishes it from these domain engineering methods and technologies.

2.3.1 DEMRAL

Domain Engineering Method for Reusable Algorithmic Libraries (DEMRAL) [CZA99a, CZA00] is a specialized domain engineering method aimed at creating a system family in order to maximize component reuse. It describes a complete analysis and design method for developing reusable libraries in algorithmic areas such as image processing, numerical computing, and containers.

Table 2.1 Outline of DEMRAL (from [CZA99a, CZA00]

- 1. Domain Analysis
  - 1.1. Domain Definition
    - 1.1.1. Goal and Stakeholder Analysis
    - 1.1.2. Domain Scoping and Context Analysis
      - 1.1.2.1. Analysis of application areas and existing systems (i.e. exemplars)
      - 1.1.2.2. Identification of domain features
      - 1.1.2.3. Identification of relationships to other domains
  - 1.2 Domain Modeling
    - 1.2.1. Identification of key concepts
    - 1.2.2. Feature modeling of the key concepts (i.e. identification of commonalities, variabilities, and feature dependencies/interactions)
- 2. Domain Design
  - 2.1. Identification of the overall implementation architecture
  - 2.2. Identification and specification of domain-specific languages
  - 2.3. Specification of the Configuration Knowledge
- 3. Domain Implementation (implementation of the domain-specific languages, language translators, and implementation components)

The development process of DEMRAL is an iterative and incremental one. The procedure of DEMRAL is outlined in Table 2.1.This method closely follows the widely accepted division of the domain engineering to divide the procedure into three phases: domain analysis, domain design and domain implementation. It was created while applying the Organization Domain Modeling (ODM) in the development of the matrix computation library [CZA00]. Although this method is not intended for generating DCS families, the procedure outline in Table 2.1 can act as a good guideline for developing a method for generating DCS families and is reflected in the USGPF.

The domain analysis in DEMRAL involves the domain definition and the domain modeling. The purpose of the domain definition is to establish the domain scope based on the analysis of stakeholders, who have interests in the ongoing project, their goals and existing systems. The purpose of the domain modeling is to model the contents of the domain by finding the relevant domain concepts and modeling their features. The domain modeling involves identification of the key concepts and the feature modeling of these concepts. By definition, DEMRAL focuses on the domains whose main concept categories are ADTs (Abstract Data Type) and algorithms. An ADT defines a whole family of data types. The feature modeling of the key concepts is to develop feature models of the concepts in the domain to define the common and variable features of the concept instances and the dependencies between variable features. The purpose of the domain design is to develop a library architecture, identify implementational components, specify domain specific languages (DSLs) constituting the application programming interface to the library, and specify the translation of the DSLs into the target architecture. The domain design builds on the results of the domain modeling and involves the following activities: scope the domain model for the implementation, identify packages, develop target architectures and identify the implementational components, identify the DSLs, identify interactions between DSLs and specify DSLs and their translation into target architectures. During the domain implementation phase, different implementational techniques, such as template meta-programming, preprocessor, compiler, and intentional programming, etc, are applied to implement different parts of an algorithmic library.

2.3.2 Draco

Draco [NEI80] began as the PhD work of James M. Neighbors. It has been used and has evolved since 1980. It is now being used to generate commercial software by Bayfront Technologies, Inc [BAY03]. Draco defines each modeling domain (such as a network domain or a database domain) by a special purpose programming language of abstractions and their operations that are specific to that domain. A modeling domain is a pure abstraction of the knowledge about the domain and makes no a priori commitment about how any operator or an abstraction in that domain will actually be implemented. The semantics of these domain specific languages are provided by a set of refinements that map the abstractions and their operations in a given domain into the abstractions and operations of other (conceptually lower and more primitive) domains. For any specific expression of operators and operands, there may be several alternative potential refinements that might apply based upon the context in which the refinement is occurring.

The basic steps in the production of a specific system using a Draco supported domain-specific high-level language is briefly described here (details can be found in Draco 1.2 Users Manual [NEI03]). For a problem domain that is understood well enough to define a domain language suitable for comfortably and easily describing systems in this domain, define the domain language and describe the domain with this language in precise meaning, provide relations among the objects and operations of the domain, and prepare a description of the meaning of the operations and objects in the domain. Specify components for the objects and operations in the domain. These components are formed into libraries. A component is a set of refinements each capable of implementing a domain object or an operation under certain stated conditions while satisfying certain implementation assertions. A new system can be described in the domain language and then turned into an internal form, which is used during the transformation and the refinement. The basic operation during the transformation and the refinement is the selection of an appropriate set of software components to implement the operations and objects in the domain which are used in the problem statement. These components then are specialized by a program transformation to the problem under consideration.

In summary, three themes dominate the way Draco operates: the use of special-purpose high-level languages for the domains or problem areas in which many similar systems are needed; the use of software components to implement problems stated in these languages in a flexible and reliable way; and the use of program transformations to tailor the components to their use in a specific context. The theory behind its operations is described in detail in Neighbors' PhD thesis [NEI80]. Although Draco is not designed for creating DCS, these three themes are not tied to creating standalone programs. In USGPF, these three themes are adopted and modified specifically for the purpose of generating DCS. USGPF also addresses issues that are not addressed by Draco, such as QoS, communication patterns and heterogeneity, etc.

2.3.2 GenVoca

GenVoca [BAT92, BAT95, BAT96, BAT02] is the distillation of the designs of two independently-conceived software system generators for the domains of databases and communications protocols. It is a tool for defining code constructs at a higher level than program code. It is a domain-independent model for defining scalable families of hierarchical systems as compositions of reusable components. The idea behind GenVoca is to compose objects out of a series of *layers*. Each layer handles a specific aspect of the object. Layers can be mixed and matched in a flexible way.

The distinguished features of GenVoca are realms, components and type equations. GenVoca defines standardized interfaces called *realm*s which may contain multiple classes and their methods. GenVoca *components* are modules that export a realm interface and encapsulate the implementation of a single design feature. GenVoca components may also import realm interfaces allowing components to be parameterized by other components. Such compositions are specified in *type equations.* Each component implements a large scale refinement; a composition of components represents a composition of such refinements. GenVoca provides techniques to decompose existing applications into reusable and composable components.

GenVoca requires the definition of standardized realm interfaces as its starting point. This is usually preceded by a domain analysis which reveals what standardized interfaces should be supported. Each interface defined represents a *subsystem* abstraction whose implementations are specified by families of subsystems (type equations), called an *application family*. More specific and detailed information about GenVoca can be found from the website: http://www.cs.utexas.edu/users/schwartz/.

In the USGPF, a set of interfaces are also created and standardized for a DCS domain, and the components need to specify the interfaces it requires and provides. The system architectures in the USGF also adopt layered architecture. Components in the USGPF are autonomous entities; however, components in GenVoca are not. GenVoca is more suitable for modeling and creating standalone programs, but USGPF is designed for DCS. The GenVoca is a domain independent model; however, the application of GenVoca to a specific domain creates a domain dependent generator. Any refinement of the domain architecture or the creation of new components requires the modification of the generator. The modification is error prone. In contrast, the proposed USGPF is domain independent and avoids this hassle. The USGPF also tries to address issues that are not considered in GevVoca, for example, QoS of components and systems, integration of heterogeneous components, etc.

This chapter provides an overview of the background and the related work that has influenced the development of the USGPF. In next chapter, an overview on the UniFrame, which is the context for the proposed USGPF, is presented.

3. OVERVIEW OF THE UNIFRAME

Chapter 2 provided an overview of the background and related work for this thesis. This chapter describes the Unified Meta-component Model Framework (UniFrame) and how it can be used for developing a DCS from a DCS family by integrating reusable heterogeneous and geographically distributed software components.

The UniFrame project is an attempt towards the unification of the existing and emerging distributed component models under a common meta-model for the purpose of enabling discovery, interoperability, and collaboration of components via generative programming techniques [RAJ00, RAJ01, RAJ02]. It specifies a framework for the component developers to create, test and verify Quality of Service (QoS) and deploy the components, and for the application programmers to select and generate a software solution for the DCS under consideration in an automatic or semi-automatic fashion (automation to the maximum possibility). The UniFrame consists of the Unified Meta-component Model (UMM) and the UniFrame Approach (UA). The Unified Meta-Component Model (UMM) proposed in [RAJ00] is the central theme of the UniFrame. UA is a component based software engineering process based on UMM for creating a DCS out of available heterogeneous and distributed software components.

This chapter provides an overview of the UMM and the UA. It also describes the implementations of various features of the UMM, including the UMM specification, the UniFrame QoS Framework (UQOS) and the UniFrame Resource Discovery Service (URDS). The last part of this chapter presents a brief discussion of the UniFrame System-Level Generative Programming Framework (USGPF), which is a framework for realizing UA at the system level. The USGPF is the theme of this thesis. The detailed descriptions about it are presented in the chapters from 4 to 7.

3.1 The Unified Meta-Component Model (UMM)

The recent shift in the focus of Object Management Group (OMG) to Model Driven Architecture (MDA) [OMG01] is a recognition that bridging components to create DCS requires standardization of not only the infrastructure but also Business and Component Models. The UMM provides an opportunity to bridge gaps that currently exist in the standards arena and provides the theoretical foundation for the UniFrame. The core parts of the UMM are: components, service and service guarantees, and infrastructure. A brief discussion of UMM is provided below. A detailed description of the UMM is available in [RAJ00, RAJ01, RAJ02].

3.1.1 Components

The UniFrame is a component-based framework. Hence, components are the building blocks of any system built by using the UniFrame. In UniFrame, components are autonomous entities with non-uniform implementations. This means that the components may adhere to different distributed computing models and there is no notion of either a centralized controller or a unified implementational framework. Every component has a state, an identity, a behavior. Thus, all components have well-defined interfaces and private implementations. In addition, each component in the UMM has three aspects: computational aspect, cooperative aspect and auxiliary aspect.

- *Computational Aspect*

  The computational aspect reflects the task(s) carried out by each component. It is a form of introspection by which every component describes its services to other components. It in turn depends upon: a) the objective(s) of the task, b) the techniques used to achieve these objectives, and c) the precise specification of the functionality offered by the component. The computational aspect of a component is described by its *inherent attributes*, which consists of simple textual information containing the book-keeping information of a component, and *functional attributes*, which consists of a formal and precise description of the computation, its associated contracts and the levels of service that the component offers.

- *Cooperative Aspect*

  The cooperative aspect of a component indicates its interactions with other components. The cooperative aspect of a component may contain: 1) *Pre-processing collaborators* - other components on which this component depends upon; and 2) *Postprocessing collaborators* - other components that may depend on this component.

- *Auxiliary Aspect*

  In addition to computation and cooperation, mobility, security, and fault tolerance are necessary features of a DCS. The auxiliary aspect of a component addresses these features.

3.1.2 Service and Service Guarantees

Services in UniFrame could be a computational effort or an access to underlying resources. In DCS, it is natural to have several choices for obtaining a specific service. Thus, each component, in addition to indicating its functionality, must be able to specify and guarantee the quality of the service offered. The quality of the service offered by a component plays an important role in whether or not the component is selected for a given system. It is an indication of a component's confidence in its ability to carry out a specified service in spite of the constantly changing execution environment and a possibility of partial failures. The QoS offered by each component is dependent upon the computation performed, algorithm used, expected computational effort and resources required, the cost of each service, and the dynamics of supply and demand.

3.1.3 Infrastructure

The headhunter [SIR02] and the Internet Component Broker (ICB) [RAJ02, SIR02] constitute the infrastructure of the UMM and allow the creation of distributed computing systems by a seamless integration of components adhering to different component models.

- Headhunter

  The headhunter is responsible for searching and managing heterogeneous and geographically distributed components. The head-hunters are analogous to binders or traders in other models. The difference is that the trader is passive, thus, the components are responsible for registering themselves with the trader. On the other hand, the head-hunter actively discovers new components and attempts to register them with itself. A component may be registered with multiple head-hunters. It is also possible for multiple head-hunters to co-operate with each other in order to discover a larger number of components.
- ICB

  The ICB is intended to act as a mediator between two components adhering to different component models. An ICB itself is a component defined under the UMM. It utilizes adapter technology to provide translation capabilities between specific component architectures. The adapter components achieve interoperability through wrap and glue technology [LUQ01]. The ICB is analogous to an Object Request Broker (ORB). The ORB provides the facilities for objects written in different programming languages to communicate, while the ICB provides the capability to generate glues and wrappers to allow components belonging to different component models to communicate.

3.2 The UniFrame Approach (UA)

The UniFrame Approach (UA) is a UMM-based technique for the automatic production of a DCS from a DCS family. The creation of a software realization of a DCS using UA has two levels: a) the component level - components are designed and developed with UMM specifications (which are informal in nature [RAJ01]), tested and validated against appropriate QoS, then deployed on the network, and b) the system level – a semi-automatic or automatic generation of a specific DCS product from a DCS family. The concepts of the generative programming are applied at both levels in the UA. This thesis describes the application of generative programming at the system level.

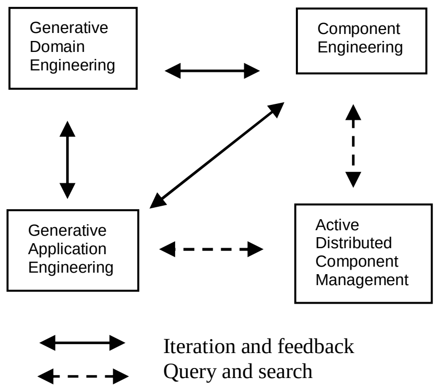


Figure 3.1 UA Core Activities

The UA has four core activities to build a DCS as shown in Figure 3.1 [HUA02]. These are: *generative domain engineering*, *component engineering*, *generative application engineering*, and *active distributed component management*. The development process is iterative and there are feedbacks during the first three activities. These four core activities span both the levels of UA: the component level and the system level. G*enerative domain engineering* and *component engineering* correspond to the domain engineering in [CZA00], aiming at maximizing the reuse of both the components and the software architecture. G*enerative domain engineering* and g*enerative application engineering* are system-level activities and the *component engineering* is at the component level. A*ctive distributed component engineerin*g is involved at both levels.

3.2.1 Generative Domain Engineering

The *generative domain engineering* consists of activities for identifying commonalities and variations of the system architecture of a DCS family to create a GDM. The GDM includes a set of abstract components as the guidelines for developing reusable concrete components during *component engineering* phase. Each abstract component represents one component type and is defined by a UMM specification. This specification is natural language-like and includes both the functional and nonfunctional (such as expected QoS properties) aspects of a component [RAJ01]. This specification is then refined into a formal specification, based upon the theory of Two-Level Grammar (TLG) [BRY02] and natural language specifications [BRY00]. This activity is the theme of the UniFrame UGDM Development Process (UGDP), which is presented in detail in Chapter 5.

3.2.2 Component Engineering

The *component engineering* phase begins with a natural language-like specification of a component. During this phase, the abstract components are mapped to different component models to create concrete components. The concrete components are tested and validated against the appropriate QoS according to the QoS Catalog [BRA01]. Then these components are deployed over the network to be discovered by the headhunters. It is worthwhile to note that the generative programming is also carried out during the *component engineering* phase.

3.2.3 Active Distributed Component Management

The a*ctive distributed component management* is the UniFrame resource discovery service (URDS) [SIR02], which is described in the section 3.5. The URDS offers the dynamic discovery and management of the heterogeneous software components and assists in the finding of the required components during the phase of the *generative application engineering*.

3.2.4 Generative Application Engineering

The g*enerative application engineering* is the process of building a DCS from a DCS family based on a GDM. This phase can be outlined in three steps: a) determining the target system and its architecture instance according to the system specification; b) searching for concrete components for the target system via the headhunters; and c) assembling and testing the DCS with the QoS validation. The GDM is used to guide this entire process. The validation of the QoS requirements is carried out both by QoS composition rules [SUN02, SUN03], which specify how the system QoS or subsystem QoS can be composed from the QoS of its parts, and by the event grammars [AUG95, AUG97], which are used as the basis for the system behavior models to trace events like executing a statement or calling a procedure. This phase is the theme of the UniFrame System Generation Infrastructure (USGI), which is presented in detail in Chapter 6.

3.3 UMM Specification

The component developers who wish to adopt the UniFrame should adhere to the UMM specification for a component and specify the parameters in the UMM specification during the component development and deployment phase. It is the responsibility of the component developer to ensure that his components meet the UMM specifications. Table 3.1 provides the UMM specification template for a component. The remaining of this section provides descriptions for each entry in Table 3.1.

Table 3.1 UMM Specification Template

UMM Specification

1. Component Name: *<component name>*
2. Component Subcase: *<component subcase name>*
3. Domain Name: *<domain name>*
4. System Name: *<system family name>*
5. Informal Description: *<natural language description>*
6. Computational Attributes:
    6.1 Inherent Attributes:
        6.1.1 id: *<internet address for a concrete component, or N/A for an abstract component>*
        6.1.2. Version: *<version expression>*
        6.1.3 Author: *<developer name for a concrete component, or N/A for an abstract component>*
        6.1.4 Date: *<deployment time for a concrete component, or N/A for an abstract component>*

(Continued in Table 3.2)

Table 3.2 UMM Specification Template (Continued from Table 3.1)

(Continued from Table 3.1)

6.1.5 Validity: <*valid time for a concrete component, or N/A for an abstract component*>

6.1.6 Atomicity: <*Yes/No*>

6.1.7 Registration: <*the registering headhunter for a concrete component, or N/A for an abstract component*>

6.1.8 Model: <*component model for a concrete component, or N/A for an abstract component*>

6.2 Functional Attributes:

6.2.1 Function description: <*natural language description of component functions*>

6.2.2 Algorithm: <*list of algorithms*>

6.2.3 Complexity: <*component complexity for a concrete component, or N/A for an abstract component*>

6.2.4 Syntactic Contract

5.2.4.1 Provided Interface: <*list of provided interfaces*>

5.2.4.2 Required Interface: <*list of required interfaces*>

6.2.5 Technology: <*technology name for a concrete component, or N/A for an abstract component*>

6.2.6 Expected Resources: <*expected resources expression, NONE if not available for a concrete component, or N/A for an abstract component*>

6.2.7 Design Patterns: <*list of used design patterns separated by comma, or NONE*>

6.2.8 Known Usage: <*list of known usage separated by semicolon, or NONE*>

6.2.9 Alias: <*list of alias separated by comma, or NONE*>

7. Cooperation Attributes:

7.1 Preprocessing Collaborators: <*list of preprocessing collaborators separated by comma or NONE*>

7.2 Postprocessing Collaborators: <*list of postprocessing collaborators separated by comma or NONE*>

8. Auxiliary Attributes:

8.1 Mobility: <*Yes/No*>

8.2 Security: <*security level* >

8.3 Fault tolerance: <*fault tolerance level* >

9. Quality of Service

9.1 QoS Metrics: <*list of QoS Metrics separated by comma for an abstract component, or list of detailed QoS Metrics separated by semicolon for a concrete component*>

9.2 QoS Level: <*level of QoS* >

9.3 Cost: <*compensation level* >

9.4 Quality Level: <*level of quality* >

- Component Name: This entry specifies the name of the component that this UMM specification is about. The name is used to identify the component.

- Component Subcase: This entry indicates information related to communication patterns of functions of the component. The communication patterns reflect the synchronization aspect of functions and are discussed in Chapter 4 and Chapter 5.
- Domain Name: This entry provides the domain scope for the component, for example, banking domain.
- System Name: This entry indicates the system family to which this component belongs to.
- Description: This entry provides an informal description of the services provided by the component. This information may include unique characteristics of the component that can not be described in other entries.
- Computational Attributes: This entry describes the computational aspect of the component in term of the following parameters.
  - Inherent Attributes:
    - ID: This is a unique string consisting of the host name and the port on which the component is running along with the name with which the component binds itself to a registry, for example: intrepid.cs.iupui.edu:8080/AccountServer.
    - Version: This entry indicates the version of the component.
    - Author: This entry indicates the authors of the component.
    - Date: This entry indicates the deployment time for a concrete component. It is not applicable for an abstract component.
    - Validity: This entry indicates whether a concrete component is valid. It is not applicable for an abstract component.
    - Atomicity: This entry indicates whether the component is atomic.
    - Registration: This entry indicates to which headhunter the component registered to. It is not applicable for an abstract component.
    - Model: This entry indicates the component model that the component adhered to.
  - Functional Attributes:

    - Function Description: This entry provides a description of each of the functions supported by the component.
    - Algorithm: This entry indicates the algorithms utilized by the component to implement its functionality if the type of the specification is *concrete component*. If the specification type is *abstract component*, then this entry means the corresponding concrete components must implement the indicated algorithms, e.g., Quick Sort.
    - Complexity: This entry describes the order of complexity of the above mentioned algorithms implemented by the component.
    - Syntactic Contract: This entry provides the computational signature of the component's service interface. The interfaces are well defined in the process of generative domain engineering. Each component must specify its provided interfaces and required interfaces.
    - Technology: This entry indicates the component technology utilized to implement the component, e.g., J2EE, CORBA, .NET etc.
    - Expected Resources: This entry indicates the expected resources for the component, e.g., CPU, memory.
    - Design Patterns: This entry indicates the design patterns employed by the component.
    - Known Usage: This entry indicates the known usages of the component.
    - Alias: This entry indicates the alias names for the component.
- Cooperation Attributes:
  - Preprocessing Collaborators: This entry indicates other components on which this component depends upon.
  - Postprocessing Collaborators: This entry indicates other components that may depend on this component.
- Auxiliary Attributes:
  - Mobility: This entry indicates whether the component is mobile or not.
  - Security: This entry indicates the security level of the component.

    - Fault Tolerance: This entry indicates the fault tolerance level of the component.
- Quality of Service:
    - QoS Metrics: Each abstract component should list the QoS metrics that should be provided by the implementation components (concrete components). For a concrete component, provided information for each QoS metrics includes: 1) QoS parameter name; 2) type of parameter: static/dynamic; 3) min/max limit. If the QoS metric is dynamic, also provide information about: 4) environment values for the min/max ratings; and 5) variation in parameter values according to environment.
    - QoS Level: A component developer may offer several possible levels of QoS. This entry is not applicable to an abstract component.
    - Cost: This entry indicates the compensation level for the component.
    - Quality Level: This entry provides an overall assessment of a concrete component. It is not applicable to an abstract component.

During the component development and deployment phase, the natural language specification is converted into a standardized XML-based specification, which can be automated discovered by the URDS.

3.4 The UniFrame QoS Framework (UQOS)

The concepts of the service and service guarantees are an integral part of every component in UMM and they also play an important role in the system generation phase of the UniFrame. The UniFrame QOS (UQOS) framework [BRA01, BRA02, BRA02a] is an implementation of the service and service guarantees aspect of the UMM.

In order to utilize the Service and Service guarantees of UMM in a real-world scenario to assure the QoS of a DCS, following issues have to be addressed: 1) a framework to objectively quantify the QoS of software components; 2) a standardized QoS Catalog for reference by software component developers and application engineers;

3) a standard approach to incorporate the effect of the environment on the QoS of software components into the component development process; 4) a standard approach to incorporate the effect of usage patterns on the QoS of software components into the component development process; and 5) a QoS specification scheme to specify the QoS of software components. The UQOS framework consists of four parts to facilitate the solving of these issues:

- *The QoS Catalog*: This catalog is intended to standardize the notion of quality of software components. It contains detailed descriptions of QoS parameters of software components, including the metrics, the evaluation methodologies, the factors influencing these parameters and the interrelationships among these parameters. In UMM, every component must specify the quality of service that it can offer in terms of the QoS parameters, as identified in the QoS Catalog.
- *The approach for accounting for the effect of the environment on the QoS of software components*: This provides methods to address the effects of diverse operating environments such as, CPU, memory, operating system and priority schemes, on the QoS of a software components. It also suggests how to document the effect in a software component so that it can be maintained by component developers and referenced by application developers.
- *The approach for accounting for the effect of usage patterns on the QoS of software components*: This consists of an empirical validation of the QoS of the software components under different usage patterns, such as the pattern of users and user requests received by components. It also suggests how to document the effect in a software component so that it can be maintained by component developers and referenced by application developers.
- *The specification of the QoS of software components*: The QoS is an integral part of every software component in the UniFrame. Thus there is a need for a formal language to specify this non-functional or QoS aspects of any software component. The specification scheme chosen for the UQOS framework is the Component Quality Modeling Language (CQML) [AAG01]. CQML is a lexical language for specifying QoS. It is based on four specification constructs, i.e., *QoS*

characteristics, *QoS statements*, *QoS profiles* and *QoS categories*. CQML meets the need of the UQOS for a generic and domain independent specification language, which can seamlessly integrate with object-oriented analysis and design, can separate the QoS specification from functional specification both syntactically and semantically, and is compatible with existing interface definition languages like CORBA IDL.

3.5 The UniFrame Resource Discovery Service (URDS)

This section provides an overview of the UniFrame Discovery Service (URDS) [SIR02], which is an implementation of the UMM infrastructure. URDS is designed to provide the infrastructure necessary for discovering and managing a collection of heterogeneous components for building a DCS. The URDS infrastructure is illustrated in Figure 3.2. The numbers in the Figure 3.2 indicate the flow of activities in the URDS. The rest of this section provides a brief description of the components of the URDS.

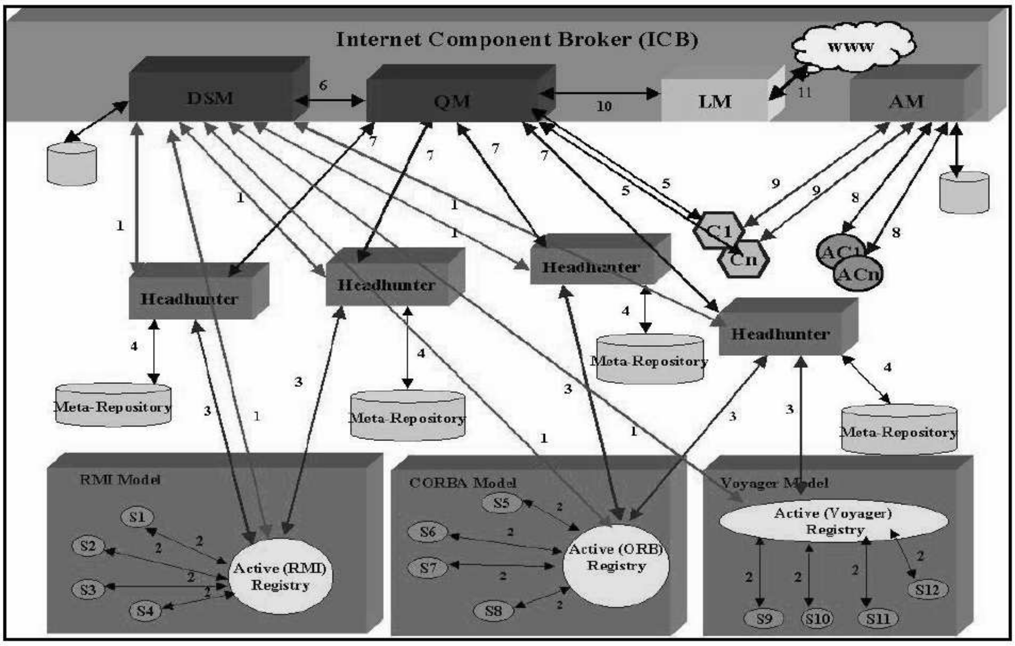


Figure 3.2 URDS Architecture (from [SIR02])

- Internet Component Broker (ICB)

  The ICB has been discussed in 3.1.3. It contains the following: Query Manager (QM), the Domain Security Manager (DSM), Link Manager (LM) and Adapter Manager (AM). The ICB acts as an all-pervasive component broker in an interconnected environment. The communication infrastructure necessary to identify and locate services, enforce domain security and handle mediation between heterogeneous components are all contained in the ICB. The services that ICB provided are accessible at well-known addresses. It is anticipated that there will be a fixed number of ICBs deployed at well-known locations hosted by organizations supporting the UniFrame Approach.

  - Query Manager (QM): The QM translates an application engineer's requirements specification for a component into a Structured Query Language (SQL) statement and dispatches this query to the appropriate head-hunters. The headhunters, in turn, return lists of components that match the search criteria contained in the query. The QM and the Link Manager together are responsible for propagating the queries to other linked ICBs.
  - Domain Security Manager (DSM): The URDS discovery protocol is based on periodic multicast announcements. The multicasting exposes the URDS to security threats. The DSM is responsible for ensuring that the security and integrity of the URDS are maintained. The security scheme implemented by the DSM involves the generation and distribution of secret keys for the ICB. It also enforces multicast group memberships and controls access to multicast addresses allocated for a particular domain.
  - Link Manager (LM): The LM establishes links between ICBs to form a federation and propagate the queries received from the QM to the linked ICBs. The ICB administrator configures the LM with the location information of LMs of other ICBs with which links are to be established.
  - Adapter Manager (AM): The AM acts as registry or lookup service for clients seeking adapter components. Adapter components register with the AM and at the

same time indicate which component models they can bridge efficiently. The AM is contacted by the clients to locate the adapter components matching their requirements.

- Headhunter (HH)

  The Headhunter has also been discussed in 3.1.3. It is responsible for the detection of the presence of service providers (service discovery), registration of functionality of the service providers and returning to the ICB a list of discovered service providers that match the requirements. Headhunters are specialized UMM components.

- Meta-Repository (MR)

  The MR is a database that serves a headhunter by holding the UMM specification information of exporters. Currently, the MR is implemented as a relational database using Oracle in the URDS.

- Active-Registries (ARs)

  The ARs listen and respond to multicast messages from headhunters. Each also has introspection capabilities to discover not only the instances, but also the specifications of the components registered with them. URDS implements them by extending the native registries or lookup services of component models like RMI, CORBA and Voyager.

- Services ($S_1..S_n$)

  The services may be implemented in diverse component models. Each identifies itself by the service type name and the XML description of the component's informal UMM specification.

- Adapter Components ($AC_1..AC_n$)

  These components serve as bridges between components implemented in different component models like (J2EE, CORBA, .NET).

Figure 3.2 also illustrates the users ($C_1..C_n$) of the URDS system who can be the Component Assemblers, System developers or System Integrators. However, in complete UniFrame, there will be no direct interaction between the human users and the URDS. The interaction would be via the interface of the system generator.

The URDS architecture is organized as a federated hierarchy as shown in Figure 3.2 in order to achieve scalability. Every ICB has zero or more Headhunters attached to it. The ICBs in turn are linked together with unidirectional links to form a directed graph. The URDS discovery process is “administratively scoped”, i.e., it locates services within an administratively defined logical domain, which refers to industry specific markets such as Financial Services, Health Care Services, Manufacturing Services, etc. The domains supported are determined by the organizations providing the URDS service. The URDS architecture is designed to handle failures through periodic announcements (in case of Headhunters), ‘heartbeat’ probes (in case of Link Managers) and information caching.

3.6 The UniFrame System-Level Generative Programming Framework (USGPF)

The QoS is an integral part of every component in UMM and is inherent in any systems generated from these components. Thus, the QoS plays an important role in the entire UniFrame Approach and helps to create QoS-aware DCS from heterogeneous distributed software components. The UniFrame Approach also shifts from the traditional software development paradigm of developing single DCS to the paradigm of developing a DCS family.

The USGPF realizes the UniFrame Approach on the system level. More specifically, it addresses the *generative domain engineering* and the *generative application engineering* aspects of the software development process in the UniFrame Approach. The USGPF is divided into three parts:

- The UniFrame Generative Domain Model (UGDM), which defines the common and variable properties of a DCS family.
- The UniFrame UGDM Development Process (UGDP), which defines the procedure to efficiently create a UGDM for a DCS family with QoS constraints.

- The UniFrame System Generation Infrastructure (USGI), which facilitates the automatic generation of QoS-aware DCS from a DCS family by integrating heterogeneous software components.

In summary, this chapter provided an overview of the UniFrame project, including the UMM, the UniFrame Approach and a brief description of the core tasks of this thesis, the USGPF. In the following chapters, the details of each part of the USGPF are presented. Chapter 4 describes the UGDM, Chapter 5 describes the UGDP, Chapter 6 describes the USGI in high level concepts, and Chapter 7 describes the design and implementation of the USGI in Java technology.

4. THE UNIFRAME GDM (UGDM)

This chapter describes the UniFrame GDM (UGDM), the first part of the UniFrame System-Level Generative Programming Framework (USGPF). The function of a UGDM is to capture the common and variable properties of a DCS family in the USGPF. Before starting the description of the UGDM, this chapter provides a brief discussion of the feature modeling and the UniFrame Domain Specific Language (UDSL), which are the tools in the USGPF that are used to model and express the UGDM stated in this chapter.

4.1 Feature Modeling

The purpose of feature modeling is to develop feature models for concepts or features in a domain. Feature models define the common and variable features of concept instances and the dependencies between the variable features. Constraints that can not be expressed in a feature diagram have to be recorded separately. In the USGPF, this is done by the constraint expression in the UDSL which is presented in detail in Section 4.2. Feature modeling [SEI03, KAN90] is a very important contribution to the domain engineering by the Software Engineering Institute (SEI) of Carnegie Mellon University and is essential to the generative programming.

As stated in [CZA00], there are two definitions of features found in domain engineering literature: 1) An end-user-visible characteristic of a system, which is the definition used in Feature-Oriented Domain Analysis (FODA); 2) A distinguishable characteristic of a concept (e.g., system, component, and so on) that is relevant to some stakeholder of the concept. The second definition is more general and is preferred by Czarnecki and Eisenecker, and is also used in the context of Organization Domain

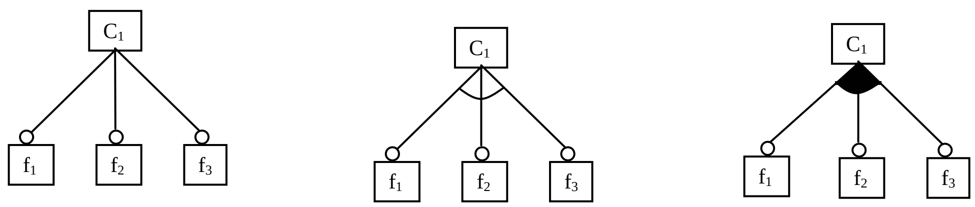
Modeling (ODM) [SIM96, SEI02b], which is also a popular domain engineering method adopted by Hewlett Packard and others. This work also adopts the second definition, because the feature modeling can be applied to any level of detail during domain engineering, which is the case in the UGDM.

Feature diagrams are usually tree-like structures, so they are also called feature trees. There are two kinds of features in feature diagrams: *mandatory* features and *optional* features. Whether a feature is mandatory or optional depends on its relationship with its parent in a feature tree. A *mandatory* feature is a feature that must be included if its parent is included in the description of a concept instance. An *optional* feature is a feature that may be included if its parent is included in the description of a concept instance. Sub-features of a feature can be grouped. There are two kinds of groups/sets: *alternative* and *or*. For an *alternative* set, if the parent of the *alternative* set is included, then exactly one feature in the *alternative* set is included in a concept instance. For an *or* set, if the parent of the *or* set is included, then any non-empty set of the *or* set can be included in a concept instance.

In the feature diagram, each feature is represented as a box. These features are then arranged in a hierarchical manner. Each feature is decomposed until it is presented at the level of interest to the users. For example, as a distributed computing system is implemented as a collection of distributed components, when modeling distributed computing system architecture in the UniFrame at the system level, the root node of a feature diagram is the target system, the inner nodes are subsystems, and the leaf nodes are abstract components, which are the blueprints for creating concrete components.

Two kinds of feature notations are used in a feature diagram to represent *mandatory* features and *optional* features respectively. A *mandatory* feature is represented by a box with a simple edge ending with a filled circle touching it. An *optional* feature is represented by a box with a simple edge ending with an open circle touching it. There are also two kinds of notations used in the feature diagram to model a grouping. An *alternative* set is represented by edges connected by an arc. An *or* set is represented by edges connected by a filled arc. See figure 4.1a for an example. The details of feature notations are described in [CZA00].

Variation points are the features which have one or more direct optional sub-features, and/or groups. In [CZA00], the authors apply the *alternative* and *or* with the mandatory features. However, this conflicts with the definition of the mandatory feature. Thus, some of the five types of variation points identified by the authors are not considered valid in UGDM. In UGDM, if a feature is mandatory, it cannot be a variable feature to its parent. Thus, only *optional* features can be used in *alternative* and *or* in the UGDM. This modification is necessary as it is consistent with the definition of *one-of* and *more-of* in the UniFrame Domain-Specific Language (UDSL). The modification does not reduce the feature diagram's ability to represent the common and variable features. It makes the semantics clearer. Figure 4.1 shows three basic variation points in the UGDM. In the figure, a) means zero or more, which can be described by *all* in the UDSL; b) means exactly one, which is an *alternative* and can be described by *one-of* in the UDSL; and c) means one or more, which is an *or* and can be described by *more-of* in the UDSL. Detail explanations are presented in the next section.



a) zero or more b) exactly one c) one or more

Figure 4.1 Types of Basic Variation Points in Feature Modeling

4.2 The UniFrame Domain Specific Language (UDSL)

The UniFrame Domain Specific Language (UDSL) is the tool in USGPF to represent the UGDM in a textual format. It is a special DSL specifically designed to be used in the USGPF. It can represent both the information contained in feature diagrams and those that can not be shown in feature diagrams. Before presenting the detail of the UDSL, a brief introduction to the DSL is presented in this section.

4.2.1 Introduction to Domain-Specific Language

A domain specific language (DSL) is a specialized, problem-oriented language. van Deursen [VAN00] provided a definition for the DSL as follows: "A domain-specific language (DSL) is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain." Domain specific language can be textual (e.g., SQL) or graphical. Well-known examples of DSLs are SQL, HTML and Make. DSLs are usually declarative. Consequently, they can be viewed as specification languages, as well as programming languages.

DSLs play an important role in generative programming because they are not only used to "order" concrete members of a system family, but also used to specify a system. The feature diagram can be expressed by a domain specific language. Feature modeling and DSL together can be used to specify a system or a family of systems to any level of detail. They can have different levels of specialization. There can be more general modeling DSLs, for example, for expression synchronization constraints, or more specialized, application-oriented DSLs. In general, several different DSLs are needed to specify a complete application. Furthermore, several different DSLs can be designed for different categories of target users to specify one single application aspect, for instance, a version for novice users and a version for advanced users.

4.2.2 Detail of the UDSL

Varghese [VAR02] modified the DSL method proposed by van Deursen and Klint [VAN02] to model and document the problem space of a domain that may involve a distributed heterogeneous environment. This modified version is adopted in this work with more modifications and is enhanced to create the UniFrame Domain Specific

Language (UDSL) to model and document the UGDM for a distributed computing domain. The UDSL in Backus-Naur Form (BNF) is summarized in Table 4.1 and Table 4.2. Keywords used in the UDSL are shown in lower case with italic font in the tables. The UDSL consists of four types of expressions to model and document a UGDM for a DCS family by the UniFrame Approach: feature expressions, constraint expressions, design feature expressions and use case expressions. These are discussed in the following sections.

Table 4.1 BNF Definition of the UDSL

UniFrame Domain Specific Language (UDSL)

<UDSL-expression> ::= <feature-expression> | <constraint-expression> | <design-feature-expression> |
<use-case-expression>

1. Feature Expression (Commonality and Variation)
<feature-expression> ::= <optional-feature> | <mandatory-feature> | <composite-feature> |
<non-exclusive-feature> | <alternative-feature>
<optional-feature> ::= <feature>*?*
<mandatory-feature> ::= <feature> | <feature>*!*
<composite-feature> ::= *all* (<feature-list>)
<non-exclusive-feature> ::= *more-of* (<optional-feature-list>)
<alternative-feature> ::= *one-of* (<optional-feature-list>)
<feature-list> ::= <mandatory-feature-list> | <optional-feature-list> |
<mandatory-feature-list>, <optional-feature-list> |
<optional-feature-list>, <mandatory-feature-list> |
<mandatory-feature-list>, <optional-feature-list>, <mandatory-feature-list> |
<optional-feature-list>, <mandatory-feature-list>, <optional-feature-list>
<mandatory-feature-list> ::= <mandatory-feature> | <mandatory-feature>,
<mandatory-feature-list>
<optional-feature-list> ::= <optional-feature> | <optional-feature>, <optional-feature-list>
<feature> ::= <atomic-feature> | <feature-expression>
<atomic-feature> ::= *FEATURE*

2. Constraint Expression
<constraint-expression> ::= <multiplicity-constraint> | <default-constraint> |
<mapping-constraint> | <satisfaction-constraint>

2.1 Multiplicity Constraint
<multiplicity-constraint> ::=
multiplicity ((<feature>, <feature>) : <multiplicity-expression>)
<multiplicity-expression> ::= *NATURAL-NUMBER* | *NATURAL-NUMBER..** |
NATURAL-NUMBER..NATURAL-NUMBER

(Continued in Table 4.2)

Table 4.2 BNF Definition of the UDSL (Continued from Table 4.1)

UniFrame Domain Specific Language (UDSL)

(Continued from Table 4.1)

2.2 Default Constraint
<default-constraint> ::= *default* (<feature>: <feature>)
2.3 Mapping Constraint
<mapping-constraint> ::= *map* (<feature> : <feature>)
2.4 Satisfaction Constraint
<satisfaction-constraint> ::= <require-constraint> | <reject-constraint> |
<mutual-require-constraint> | <include-constraint> | <exclude-constraint>
<require-constraint> ::= *require* (<feature-list>)
<reject-constraint> ::= *reject* (<feature-list>)
<mutual-require-constraint> ::= *mutual_require*(<feature-list>)
<include-constraint> ::= *include* (<feature>, <feature>)
<exclude-constraint> ::= *exclude* (<feature>, <feature>)

3. Design Feature Expression
<design-feature-expression> ::= <design-feature-interaction> | <design-feature-interface>
<design-feature-interaction> ::= *interact* (<design-feature>, <design-feature>)
<design-feature-interface> ::= *interface* (<design-feature>: *provided_interface* (<interface-list>), *required_interface* (<interface-list>))
<interface-list> ::= *INTERFACE* | *INTERFACE*, <interface-list>
<design-feature> ::= *SYSTEM* | *SUBSYSTEM* | *ABSTRACT-COMPONENT*

4. Use Case Expression
<use-case-expression> ::= <use-case-component-level> | <use-case-function-level>
<use-case-component-level> ::= *USE-CASE* : *path_c* (<abstract-component-list>)
<use-case-function-level> ::= *USE-CASE* : *path_f* (<function-call-list>)
<abstract-component-list> ::= *ABSTRACT-COMPONENT* | *ABSTRACT-COMPONENT*,
<abstract-component-list>
<function-call-list> ::= <function-call> | <function-call>, <function-call-list>
<function-call> ::= *ABSTRACT-COMPONENT.FUNCTION*[<communication-pattern>]
<communication-pattern> ::= *cp1* | *cp2s* | *cp2a*

The UDSL is used to express not only the feature diagrams, but also other models in the UGDM, such as the Critical Use Case Model (CUCM), QoS Composition and Decomposition Model (QCDM), etc. Models expressed by the UDSL can be further formally expressed by two-level grammar (TLG)[BRY02]. The work on using TLG to formally express the UGDM is underway at University of Birmingham, a collaborator of the UniFrame research.

<u>4.2.2.1 Feature Expressions</u>

The feature expression is used to express the commonality and variation of a feature diagram in the UGDM. There are five types of feature expressions: optional feature, mandatory feature, composite feature, non-exclusive feature and alternative feature.

- <optional-feature> ::= <feature>*?*

  An optional feature is expressed as a feature followed by a question mark. An optional feature means this feature may be included if its parent is included in the description of a concept instance. An example of an optional feature is presented in the description of the composite feature below.

- <mandatory-feature> ::= <feature> | <feature>*!*

  A mandatory feature is expressed as a feature followed by an exclamation mark. The exclamation mark can be omitted. A mandatory feature means this feature must be included if its parent is included in the description of a concept instance. An example of a mandatory feature is presented in the description of the composite feature below.

- <composite-feature> ::= *all* (<feature-list>)

  A composite feature is composed from features in the feature list. It is defined as a feature list preceded by the *all* keyword. Features in this feature list can be optional, mandatory, or a mixture of both. If all the features in this feature list are optional, it represents the "zero or more" variation point as shown in Figure 4.1. If the feature list consists of only one feature, then the keyword *all* can be omitted. For example, in the expression, *UserSubsystem : all (ATM?, CashierTerminal),* the composite feature *UserSubsystem* is composed from *ATM* which is an optional feature and *CashierTerminal* which is a mandatory feature. In this example, *Usersubsystem* is the parent of *ATM* and *CashierTerminal,* thus if UserSubsystem is included in a system description, CashierTerminal must be present; however, ATM may be present.

- <non-exclusive-feature> ::= *more-of* (<optional-feature-list>)

  A non-exclusive feature is defined as an optional feature list preceded by the *more-of* keyword. Every feature in the feature list may be present; however, there must be at least one feature to be present if the non-exclusive feature is present. It expresses the "one or more" variation point as shown in Figure 4.1. It has the meaning of *or* in a feature model. If the feature list consists of only one feature, then the keyword *more-of* can be omitted. For example, in the expression, *TransactionServerSubsystem: more-of (DeluxeTransactionServer, EconomicTransactionServer)*, the non-exclusive feature *TransactionServerSubsystem* is defined by the optional feature list *(DeluxeTransactionServer, EconomicTransactionServer)*; thus, it can be either one of the two optional features in the optional feature list or can be both of them, i.e., there are three possibilities for *TransactionServerSubsystem*.

- <alternative-feature> ::= *one-of* (<optional-feature-list>)

  An alternative feature is defined as an optional feature list preceded by the *one-of* keyword. It expressed the "exactly one" variation point as shown in Figure 4.1. It has the meaning of *alternative* in a feature model. If the feature list consists of only one feature, then the keyword *one-of* can be omitted. For example, in the expression, *IAccountDatabase: one-of (IAccountDatabase1, IAccountDatabase2)*, the alternative feature *IAccountDatabase* is defined as either *IAccountDatabase1* or *IAccountDatabase2*.

In the UDSL, a feature list is a list of features without any ordering constraints. Features in a feature list can be *optional* or *mandatory*. Thus, a feature list may consist of only *optional* features, only *mandatory* features or a mixture of both in any order. An *optional* feature list is a special feature list in which all features are *optional*. A mandatory feature list is another special feature list in which all features are *mandatory*. Since the feature lists in non-exclusive features and *alternative* features are all optional feature lists, the question mark can be omitted for the *optional* features in these lists.

4.2.2.2 Constraint Expressions

Constraints reveal the relations that cannot be deduced from feature expressions. It further limits the variability of a feature diagram. In the UDSL, there are four categories of constraint expressions: multiplicity constraint, default constraint, mapping constraint and satisfaction constraint.

4.2.2.2.1 Multiplicity Constraint

The multiplicity constraint reveals the multiplicity relationship between different features. The syntax for this constraint expression is defined as:

- <multiplicity-constraint> ::=
  multiplicity ((<feature>, <feature>) : <multiplicity-expression>)
  The keyword for the multiplicity expression is *multiplicity*. The expression follows the UML convention in expressing multiplicity. The multiplicity for the first feature is 1 and the one for the second feature is indicated by the multiplicity expression. The meaning is for one instance of the first feature, how many instances of the second feature are related. For example, in the expression, *multiplicity ((Bank, TransactionServerManager) : 1),* each *Bank* is related to one copy of *TransactionServerManager*.

The multiplicity expression used in the multiplicity constraint includes: 1) *NATURAL-NUMBER*, which is any non-negative integer; 2) *NATURAL-NUMBER..**, which means at least the number specified by *NATURAL-NUMBER*. Two special cases are 0..*, which means zero or more, and 1..*, which means one or more; 3) *NATURAL-NUMBER..NATURAL-NUMBER,* which denotes the range specified by the two *NATURAL-NUMBER* in the expression, the second one of which must be larger. Other multiplicity expressions can also be defined when needed.

4.2.2.2.2 Default Constraint

The default constraint is used to express the default value in a feature expression. There is only one expression in this category. The syntax for this expression is defined as:

- <default-constraint> ::= *default* (<feature>: <feature>)
  The keyword for the default expression is *default*. The expression means the default feature for the first feature is the second feature in the expression. The first feature is usually a variation point and the second feature is a sub-feature of the first feature. For example, in the expression, *default (UserSubsystem: CashierTerminal),*

UserSubsystem is a variation point as shown in the example of composite feature, the default for it is *CashierTerminal*.

4.2.2.2.3 Mapping Constraint

The mapping constraint is used for mapping from one model to another model in the UGDM. The mapping can be considered as a kind of transformation. It can relate different models, or it can reveal more detailed lower level information from one model to anther in a hierarchical setting. Examples of mapping are the architecture to critical use case model mapping at function/interface level and the architecture model mapping, which are shown in Section 4.3.

- <mapping-constraint> ::= *map* (<feature>, <feature>)

  The keyword for the mapping constraint is *map*. The expression means that the first feature is mapped to the second feature in the expression. The mapping is not reversible, i.e., it is not symmetric. However, the mapping is transitive. For example, in the expression, *map (BankCase1: BankCase1_1)*, *BankCase1* is an architecture instance at the abstract component level, *BankCase1_1* is an architecture instance at the function/interface level, and *BankCase1* is mapped to *BankCase1_1*. *BankCase1_1* consists of more detailed information than *BankCase1* about the system architecture and the reverse mapping loses information, which will become clear in Chapter 5. That is why mapping constraint is not reversible.

4.2.2.2.4 Satisfaction Constraint

The satisfaction constraint reflects the constraints identified in van Deursen and Klint’s work [VAN02] with slight modifications and some enhancements. The satisfaction constraint includes five types of constraints in the UDSL: *require constraint*, *reject constraint*, *include constraint*, *exclude constraint* and *mutual_require constraint*. The first four types have the same semantics as requires, excludes, include and exclude respectively in van Deursen’s work. The *mutual_require constraint* is added to the satisfaction constraints in the UDSL to simplify the expression of the situation in that a list of features must all be present or none is present. In [VAN02], the satisfaction constraints are classified into two categories: diagram constraints and user constraints. The diagram constraints express fixed and inherent dependencies across features in a feature model. The user constraints express the user requirements regarding the presence or absence of a feature, thus, it is used in the application engineering to specify system requirements. The diagram constraints consist of *require constraint*, *reject constraint* and *mutual_require* constraint. The user constraints consist of *include constraint* and *exclude constraint*. Following is the syntax and brief description of each satisfaction constraint.

- <require-constraint> ::= *require* (<feature>, <feature>)

  This satisfaction rule expresses the constraint that if the first feature is present, then the second feature must be present as well. The keyword to express this kind of constraint is *require*. For example, the expression, *require (SavingAccount, InterestRate)*, means *SavingAccount* is associated with *InterestRate*. However, the reverse might not be true. For example, *InterestRate* can be associated with *MoneyMarketAccount*, not *SavingAccount*.
- <reject-constraint> ::= *reject* (<feature>, <feature>)

  This satisfaction rule expresses the constraint that if the first feature is present, then the second feature must not be present. The keyword to express this kind of constraint is *reject*. Fore example, the expression, *reject (CheckingAccount, InterestRate)*, means *CheckingAccount* can not be associated with *InterestRate*. The reverse is also true in this rule. *InterestRate* can not be associated with *CheckingAccount*.
- <mutual-require-constraint> ::= *mutual_require* (<feature list>)

  This satisfaction rule expresses the constraint that if any feature in the feature list is present, all other features in the feature list must be present as well. There can be more than two features in the expression. The keyword to express this kind of constraint is *mutual_require*. For example, the expression, *mutual_require (ATM, CustomerValidationServer)*, means *ATM* and *CustomerValidationServer* must be present together, or none of them is present.

- <include-constraint> ::= *include* (<feature list>)

  This satisfaction rule expresses the constraint set by users that the features included in the feature list must be present in a generated system to satisfy the system requirement. The keyword to express this kind of constraint is *include*. For example, the expression, *include (ATM)*, means the user requires the generated system to contain *ATM*.
- <exclude-constraint> ::= *exclude* (<feature list>)

  This satisfaction rule expresses the constraint set by users that features in the feature list must not be present in a generated system. The keyword to express this kind of constraint is *exclude*. For example, the expression, *exclude (ATM)*, means the user requires the generated system must not contain *ATM*.

4.2.2.3 Design Feature Expressions

Design features are used to capture a hierarchical system architecture in the UGDM. A hierarchical system architecture in the USGPF is formed into layers. Elements in layers are classified into three types that are captured as design features: *system*, *subsystem* and *abstract component*. The root of a system architecture hierarchy is a design feature of *system*. The leaves of a system architecture are design features of *abstract component*. The rest of a system architecture are design features of *subsystem*. The rationale is that a system is composed from a set of subsystems, a subsystem is composed from a set of abstract components, and abstract components are the building blocks. Thus a subsystem can be viewed as a composite component. The detail about this hierarchical architecture is described in Chapter 5. The purpose of design feature expressions is to capture the interfaces of design features and the interactions between design features. An interface here is defined as a set of published functionality available to public invocation.

- <design-feature-interaction> ::= *interact* (<design-feature>, <design-feature>)

  This statement says the first design feature interacts with the second design feature. The first feature is the initiator of this interaction and the second feature is the responder of this interaction. The interaction expressed in this rules reflect the cooperative aspect of components in the UniFrame. The first design feature is the preprocessing collaborator of the second design feature and the second design feature is the post-processing collaborator of the first design feature. If both design features can be the initiator of the interaction, that is, they are peers; then, two statements of design feature interaction are required to capture this kind of interaction. One special case of this rule is that the first design feature can be users. Thus, it can be extended to describe user-system interactions. The keyword for this expression is *interact*. For example, the expression, *interact (CashierTerminal, CashierValidationServer),* reveals the interaction between *CashierTerminal* and *CashierValidationServer*, and *CashierTerminal* is the initiator of the interaction.

- <design-feature-interface> ::= *interface* (<design feature>: *provided_interface* (<interface-list>), *required_interface* (<interface-list>))

  This statement expresses the provided interfaces and required interfaces for a design feature. The provided interfaces are those interfaces provided by a design feature to other design features. The required interfaces are those interfaces required by this design feature from other design features. The interfaces in the interface-list are defined for a domain during UGDP. Detail of how to develop these interfaces is presented in Chapter 5. For any design feature, it must provide an interface for other design features. However, it may not require any interfaces from any other design feature. Thus, the interface list for the required interface may be empty. When it is empty, denote it as NONE. There are three keywords in this expression to achieve the necessary semantics: *interface*, *provided_interface* and *required_interface*. For example, the expression, *interface (DeluxeTransaxtionServer: provided_interface (IAccountManagement, ICustomerManagement), required_interface (IAccountDatabase))*, states that the provided interfaces for *DeluxeTransaxtionServer* are *IAccountManagement* and *ICustomerManagement*, and the required interface for *DeluxeTransaxtionServer* is *IAccountDatabase*.

4.2.2.4 Use Case Expressions

A use case expression captures the realization of a use case in a sequence diagram. It is an ordered sequence of abstract components or function calls, depending on the level of detail. A use case can be described at two levels, abstract component level and function/interface level as indicated by the two use case expressions.

- <use-case-component-level> ::= <use-case> : *path_c* (<abstract-component-list>)

  This statement describes the expression for a use case at the abstract component level. At the abstract component level, a use case is described by a set of abstract components. An ordered sequence of interactions of these abstract components realizes the use case. The first abstract component in the list is the initiator of this use case. For example, the expression, *DepositMoneyCase1: path_c (CashierTerminal, DeluxeTransactionServer, AccountDatabase),* means the use case *DepositMoneyCase1* is realized by the cooperation of following three components in order: *CashierTerminal*, *DeluxeTransactionServer*, and *AccountDatabase*, i.e, *CashierTermianal* communicates with *DeluxeTransactionServer*, which then communicates with *AccountDatabase*. Angular brackets are used to enforce another order constraint. For example, *OpenAccountCase2: path_c (<CashierTerminal, TransactionServerManager>, EconomicTransactionServer),* means the use case *OpenAccountCase2* is realized by the cooperation of the three components in the following way: *CashierTerminal* firstly communicates with *TransactionServerManager*, then it communicates with *EconomicTransactionServer*.
- <use-case-function-level> ::= <use-case> *path_f* (<function-call-list>)

  This statement describes the path for a use case at the function/interface level. At the function/interface level, the use case is described by a set of function calls. An ordered sequence of function calls realizes the use case. The first function call in the list is the initiating function call of this use case. The syntax for a function call is presented next in this section. For example, the expression, *DepositMoneyCase1_1:* path_f (*CashierTerminal.deposit[cp2s], DeluxeTransactionServer.deposit[cp2s], AccountDatabase.getAccount[cp2s], AccountDatabase.saveAccount[cp2s]),* reveals the ordered sequence of function calls that realizes the use case *DepositMoneyCase1_1*.

- <function-call> ::= <abstract-component>.<function>[<communication-pattern>]

  This provides the syntax for a function call in the UGDM. A function call is an interaction between two components. An initiator component calls a function provided by a responder. The syntax specifies the abstract component that provided the function, the name of the function and the associated communication pattern for the function. The communication pattern provides information about parallelism, i.e. the synchronization aspect of function calls. The basic communication patterns considered are:

  - *one way*: This communication pattern is denoted as *cp1*. It describes the situation in which an initiator initiates an interaction but it does not expect any response from a responder. Thus the initiator calls the responder and then continues to do its work.
  - *two way synchronous*: This communication pattern is denoted as *cp2s*. It describes the situation in which an initiator initiates an interaction and waits until it receives a response from the responder before it can do anything else.
  - *two way asynchronous*: This communication pattern is denoted as *cp2a*. It describes the situation in which an initiator initiates an interaction and expects a response from the responder. However, it does not wait for the response. Instead, it continues to do other things. When the response comes, then it reacts to the response. Thus, the communication happens in an asynchronous manner.

4.2.3 Three Forms of the Feature Description for a Feature Diagram in the UDSL

As stated above, a feature diagram can be expressed using the UDSL. The feature description for a feature diagram in the UDSL is organized into three forms: hierarchical form, normalized form and disjunctive normal form. The hierarchical form is the direct textual description of a feature diagram. Given a direct textual representation of a feature diagram, further operations like normalization and expansion can be applied to transform the hierarchical form into other forms. The normalization rules and expansion rules stated in [VAN02] are adopted in this work and are shown in Appendix A and each rule is followed by a simple description. Examples of applying these rules on feature diagrams are developed in Chapter 5. The possible implementation of these rules and the constraint checking during applying these rules by Generic Modeling Environment [GME] are discussed in Chapter 8. These transformations are important because they are the process to produce distinctive and customized instances from a system family described by a set of feature diagrams with common and variable properties.

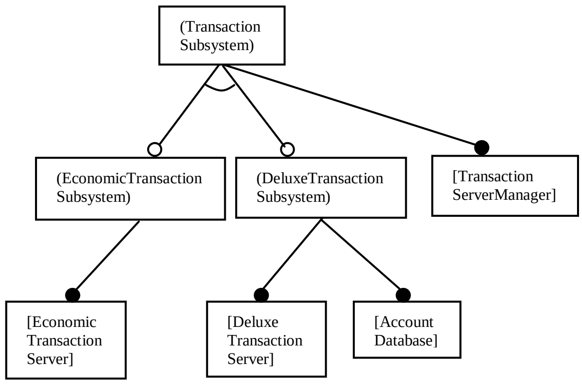


Figure 4.2 Feature Diagram of *TransactionSubsystem* in the Banking Domain Example

Table 4.3 Feature Description of *TransactionSubsystem* in the Hierarchical Form

An Example of Feature Description in the Hierarchical Form
TransactionSubsystem: all (TransactionServerManager, one-of (EconomicTransactionSubsystem, DeluxeTransactionSubsystem)) EconomicTransactionSubsystem: EconomicTransactionServer DeluxeTransactionSubsystem: all (DeluxeTransactionServer, AccountDatabase)

4.2.3.1 Hierarchical Form

The hierarchical form is the direct description of a feature diagram as a textual expression. It expresses the feature diagram from the root to the leaves. The first statement in the hierarchical form is the expression for the root feature (or concept) in terms of all its direct children in the feature diagram. The rest of statements in the hierarchical form describe inner features in the feature diagram in terms of their direct children. Figure 4.2 shows the feature diagram for the *TransactionSubsystem* developed in Chapter 5 for the banking domain example. Table 4.3 shows the description of this feature diagram in hierarchical form in the UDSL.

4.2.3.2 Normalized Form

The hierarchical form can be transformed into normalized form, which expresses the root feature (or concept) in terms of the leave features without the inner features in a feature diagram. The way to transform a hierarchical form into a normalized form is simple: for the root expression in the hierarchical form, substitute features in the right hand side of the expression until all the features in the right hand side are leaf features; then apply the normalization rules as described in van Deursen's work [VAN02]. The purpose of the normalization is to simplify the feature expression by removing the duplicate features and restructuring the expression. Table 4.4 shows an example of feature description in the normalized form derived from the hierarchical form shown in Table 4.3 for *TransactionSubsystem*.

Table 4.4 Feature Description of *TransactionSubsystem* in the Normalized Form

An Example of Feature Description in the Normalized Form
TransactionSubsystem: all (TransactionServerManager, one-of (EconomicTransactionServer, all (DeluxeTransactionServer, AccountDatabase)))

4.2.3.3 Disjunctive Normal Form

The normalized form can be further transformed into disjunctive normal form, which is defined as follows in van Deursen and Klint's work [VAN02]:

$$\textit{one-of}\ (\ \textit{all}\ (\mathrm{A}_{11}, \ldots, \mathrm{A}_{1(n1)}), \ldots, \textit{all}\ (\mathrm{A}_{m1}, \ldots, \mathrm{A}_{m(nm)}))$$

The outermost operator of a disjunctive normal form is *one-of*, and its arguments are all *all*s with arguments of only mandatory feature lists containing leave features. The resulting representation is essentially a list of all possible configurations. This transformation is done by the expansion rules described in van Deursen and Klint's work [VAN02]. During this transformation, each disjunct is checked against the appropriate constraints to determine whether the disjunct is valid or not. Table 4.5 shows an example of a feature description in the disjunctive normal form derived from the normalized form shown in Table 4.4 for *TransactionSubsystem*. This example shows two disjuncts for *TransactionSubsystem*, which means two designs.

Table 4.5 Feature Description of *TransactionSubsystem*
in the Disjunctive Normal Form

An Example of Feature Description in the Disjunctive Normal Form
TransactionSubsystem: one-of (all (TransactionServerManager, EconomicTransactionServer), all(TransactionServerManager, DeluxeTransactionServer, AccountDatabase))

4.2.4 Implementation of the UDSL

Many DSLs are supported by a DSL compiler which generates applications from a DSL program. In this case, the DSL compiler is referred to as an application generator in the literature [CLE88]. A DSL is usually implemented in two steps: 1) firstly, construct a library that implements the semantic notations; 2) secondly, design and implement a compiler that translates DSL programs to a sequence of library calls. The UDSL is implemented in an analogous way in the USGPF. Firstly, the UGDM described by the UDSL is constructed into a UGDM Knowledge Base (UGDMKB), which can consist of both databases and libraries. Secondly, a system generator which consists of the processing logics of the UGDM is designed and implemented. This will become clear in the latter chapters.

4.3 The UniFrame GDM (UGDM)

The outline for the UGDM is shown in Table 4.3. The UGDM consists of three parts: general information, which includes a description for the domain modeled; a problem space, which an application programmer can use to specify the needs; and a solution space, which contains various models including configuration knowledge to provide solutions for a DCS family. The detailed description of the UGDM is in the coming sections with examples from a banking domain developed in Chapter 5. The complete UGDM for the banking domain example is provided in Appendix J.

Table 4.6 Outline of the UGDM

Outline of the UGDM

1. General Information
    1.1 Domain Name
    1.2 System Family Name
    1.3 Version
    1.4 Date
    1.5 Author
    1.6 Description
2. Problem Space
    2.1 Use Case Model
    2.2 QoS Requirement Model
    2.3 Architecture Model in Hierarchical Form
    2.4 System-Level Multiplicity Model
3. Solution Space
    3.1 Architecture-Related Models
        3.1.1 Architecture Model in Disjunctive Normal Form (Abstract Component Level)
        3.1.2 Architecture Model in Disjunctive Normal Form (Function/Interface Level)
        3.1.3 Architecture Model Mapping
        3.1.4 Abstract Component Interaction Model
        3.1.5 Component-Level Multiplicity Model
    3.2 Design-Feature-Related Models
        3.2.1 Interface Model
        3.2.2 Abstract Component Interface Model
        3.2.3 Abstract Component Model
    3.3 QoS-Related Models
        3.3.1 Critical Use Case Model (Function/Interface Level)
        3.3.2 Architecture Model in Disjunctive Normal Form and Critical Use Case Model Mapping (Function/Interface Level)
        3.3.3 QoS Composition and Decomposition Model (QCDM)

4.3.1. General Information in the UGDM

This section of the UGDM provides the general information about a DCS family that is captured and modeled by the UGDM. The general information includes a domain name, a system family name, a version number, a creating date, authors and an informal brief description for the DCS family.

- *Domain Name*: This entry describes the full name of a domain. Domain names are organized into a hierarchical structure for the simplicity. The root of the hierarchy is represented as /. The root consists of multiple top domains, such as finance, transportation, communication, etc. Each top domain consists of multiple sub-domains. For example, the finance domain may be divided into insurance, banking, mortgage, etc. The separator between a domain or a sub-domain and its sub-domains is also /. Thus the name looks like an absolute path name of a file. For example, the name for the banking domain can be */Finance/Banking*.
- *System Family Name*: This entry describes the name of the system family in a DCS domain that this UGDM models. This can also be viewed as a sub-domain for the domain indicated in the above entry. However, they are different. Systems are standalone concrete entities in the world, but domains and sub-domains in the above entry are abstract higher level concepts. For example, the system family name for the example developed in Chapter 5 is *Bank*, which represents the real world system. It is from the *Banking* sub-domain in the *Finance* domain. *Finance* and *Banking* are higher abstract concepts.
- *Version*: This entry documents the version of the UGDM for a domain. As the development of a UGDM for a domain is an iterative and incremental process, a UGDM for a domain evolves over time. The notion of version is the way to track the history.
- *Date*: This entry documents when this UGDM was developed.
- *Author*: This entry shows the developers, maintainers or the responsible organizations for this UGDM.

- Description: This entry provides an informal text description for the system family to be described in this UGDM. This includes special information or characteristics that can not be captured by other entries of the UGDM.

4.3.2 Problem Space in the UGDM

This section of the UGDM consists of three models: Use Case Model (UCM), QoS Requirement Model (QRM) and Architecture Model in Hierarchical Form (AMHF). These models provide problem related domain specific concepts and features. Information provided by these models can be used to express an "order" for a system from a DCS family by users (system integrators, or application engineers).

Table 4.7 An Example of UCM

Use Case Model of the Banking Domain Example

1. Commonality and Variation
    - Bank: all (ManageCustomers, ManageAccounts, Login-exitAccount, ValidateUsers)
    - ManageCustomers: all (OpenAccount, CloseAccount)
    - ManageAccounts: all (ManageAccounts_Cashier, ManageAccounts_Customer?)
    - ManageAccounts_Cashier: all (WithdrawMoney_Cashier, DepositMoney_Cashier, TransferMoney_Cashier, CheckBalance_Cashier)
    - ManageAccounts_Customer: all (WithdrawMoney_Customer, DepositMoney_Customer, TransferMoney_Customer, CheckBalance_Customer)
    - ValidateUsers: all (ValidateUsers_Cashier, ValidateUsers_Customer?)
    - Login-exitAccount: all (Login-exitAccount_Cashier, Login-exitAccount_Customer?)

2. Constraint Expression
    - 2.1 Default Constraint
        - default (ManageAccounts: ManageAccounts_Cashier)
        - default (ValidateUsers: ValidateUsers_Cashier)
        - default (Login-exitAccount: Login-exitAccount_Cashier)
    - 2.2 Satisfaction Constraint
        - mutual_require (ValidateUsers_Customer, ManageAccounts_Customer, Login-exitAccount_Customer)

4.3.2.1 Use Case Model (UCM)

This model provides the information about the domain requirements highlighting the necessary functional aspects. Use cases describe externally visible behaviors of a system. This model describes the common and variable requirements for a product line in a domain. The UGDP is a use case driven process. The use case model is an essential artifact in this process. Table 4.7 shows an example of the UCM for the banking domain developed in Chapter 5. The UCM consists of two parts. The first part is the description of the commonalities and variations of the use cases for a DCS family. The second part provides the constraints between use cases.

4.3.2.2 QoS Requirement Model (QRM)

This model provides the information about the domain requirements highlighting the non-functional aspects, namely, the QoS aspects, which is an inherent characteristic of UniFrame. It is important to identify and model the domain QoS requirement in order to build QoS-aware DCS. The QRM can be viewed as the QoS aspect at the system level that can be used to express system QoS requirements. For example, in Table 4.8, the QRM states that the QoS aspect of the bank DCS family is described by system throughput and system end to end delay, which are derived from critical use case models. More detail about this model can be found in Section 5.2.2.3.

Table 4.8 An Example of the QRM

QoS Requirement Model of the Banking Domain Example
System.QoS: all (System.QoS.throughput, System.QoS.endToEndDelay) System.QoS.throughput: CriticalUseCaseModel.QoS.thoughput SystemQoS.endToEndDelay: CriticalUseCaseModel.QoS.endToEndDelay

4.3.2.3 Architecture Model in Hierarchical Form (AMHF)

This model provides the information about the domain requirements highlighting the architectural aspects of a DCS family. It reflects the commonality and variation in the architecture of a DCS family. The hierarchical form is a layered design, which is described in detail in Section 5.3.1. Table 4.9 shows the example of the AMHF from a banking domain developed in Chapter 5. The AMHF consists of two parts. The first part is the description of the commonalities and variations of the architecture for a DCS family. The second part provides the constraints between architecture properties.

Table 4.9 An Example of the AMHF

Architecture Model in Hierarchical Form of the Banking Domain Example
1. Commonality and Variation Bank: all (UserSubsystem, UserValidationSubsystem, TransactionSubsystem) UserSubsystem: all (ATM?, CashierTerminal) UserValidationSubsystem: all (CustomerValidationServer?, CashierValidationServer) TransactionSubsystem: all (TransactionServerManager, one-of (EconomicTransactionSubsystem, DeluxeTransactionSubsystem)) EconomicTransactionSubsystem: EconomicTransactionServer DeluxeTransactionSubsystem: all (DeluxeTransactionServer, AccountDatabase)
2. Constraint Expression 2.1 Default Constraint default (UserSubsystem: CashierTerminal) default (UserValidationSubsystem: CashierValidationServer) default (TransactionSubsystem: all (TransactionServerManager, EconomicTransactionSubsystem) 2.2 Satisfaction Constraint mutual_require (ATM, CustomerValiationServer)

Table 4.10 An Example of the System-Level MM

System-Level Multiplicity Model of the Banking Domain Example
multiplicity ((Bank, CashierTerminal): 1..*) multiplicity ((Bank, ATM) : 0..*) multiplicity ((Bank, CashierValidationServer) : 1) multiplicity ((Bank, CustomerValidationServer) : 0..1) multiplicity ((Bank, TransactionServerManager) : 1) multiplicity ((Bank, EconomicTransactionServer) : 0..2) multiplicity ((Bank, DeluxeTransactionServer) : 0..2) multiplicity ((Bank, AccountDatabase) : 0..2)

4.3.2.4 System-Level Multiplicity Model (MM)

The multiplicity model defines the multiplicity constraints in a system. The multiplicity constraints are defined at two levels: system-level multiplicity and component-level multiplicity. The system-level multiplicity expresses the multiplicity of the root feature (a system) in terms of leaves (abstract components) in a feature diagram of AMHF. The component-level multiplicity is a solution space topic and is discussed in the Section 4.3.3.1.5.

4.3.3 Solution Space in the UGDM

This section presents various models including the configuration knowledge to provide solutions for a DCS family. These models are organized into three categories: architecture related models, design feature related models and QoS related models. Some configuration knowledge is reflected in models, such as the Architecture Model in Disjunctive Normal Form (AMDNF) reflects the configuration knowledge of illegal component combinations when it is transformed from Architecture Model in Normalized Form (AMNF) by the expansion rules. More details are presented in Chapter 5.

4.3.3.1 Architecture-Related Models

The UGDM provides a common architecture with variations for a DCS family. The commonalities and variations in the architecture are reflected in different architecture related models, such as Architecture Model in Disjunctive Normal Form (AMDNF) at both abstract component level and function/interface level, Architecture Model Mapping (AMM), Abstract Component Interaction Model (ACIM) and component-level Multiplicity Model (MM). These models also reflect the architecture at different level of details.

4.3.3.1.1 Architecture Model in Disjunctive Normal Form (Abstract Component Level)

The Architecture Model in Disjunctive Normal Form (AMDNF) at the abstract component level shows what kind of abstract components are needed for an architecture instance at the component level without concerning about the lower level of detail like communication patterns. This model is derived from the AMHF by normalization and expansion as discussed in Section 4.2.3. Table 4.11 provides an example of the AMDNF at the abstract component level from the banking domain.

Table 4.11 An Example of the AMDNF at the Abstract Component Level

AMDNF at Abstract Component Level for the Banking Domain Example
1. Disjunctive Normal Form Bank: one-of (BankCase1, BankCase2, BankCase3, BankCase4) BankCase1: all (ATM, CashierTerminal, CustomerValidationServer, CashierValidationServer, TransactionServerManager, EconomicTransactionServer) BankCase2: all (ATM, CashierTerminal, CustomerValidationServer, CashierValidationServer, TransactionServerManager, DeluxeTransactionServer, AccountDatabase) BankCase3: all (CashierTerminal, CashierValidationServer, TransactionServerManager, EconomicTransactionServer) BankCase4: all (CashierTerminal, CashierValidationServer, TransactionServerManager, DeluxeTransactionServer, AccountDatabase) 2. Constraint Expression 2.1 Default Constraint Default (Bank: BankCase3)

4.3.3.1.2 Architecture Model in Disjunctive Normal Form (Function/Interface Level)

The Architecture Model in Disjunctive Normal Form (AMDNF) at the function/interface level provides all possible architecture instances for a DCS family. It also provides more detail information such as the necessary communication patterns in an architecture instance. Table 4.12 provides an example of the AMDNF at the function/interface level from the banking domain.

Table 4.12 An Example of the AMDNF at the Function/Interface Level

```
AMDNF at Function/Interface Level for the Banking Domain Example

1. Disjunctive Normal Form
   Bank: one-of (BankCase1, BankCase2, BankCase3, BankCase4)
   BankCase1: one-of (BankCase1_1)
   BankCase2: one-of (BankCase2_1, BankCase2_2)
   BankCase3: one-of (BankCase3_1)
   BankCase4: one-of (BankCase4_1, BankCase4_2)
   BankCase1_1: all (ATMCase1, CashierTerminalCase1, CustomerValidationServerCase1,
       CashierValidationServerCase1, TransactionServerManagerCase1,
       EconomicTransactionServerCase1)
   BankCase2_1: all (ATMCase1, CashierTerminalCase1, CustomerValidationServerCase1,
       CashierValidationServerCase1, TransactionServerManagerCase1,
       DeluxeTransactionServerCase1, AccountDatabaseCase1)
   BankCase2_2: all (ATMCase1, CashierTerminalCase1, CustomerValidationServerCase1,
       CashierValidationServerCase1, TransactionServerManagerCase1,
       DeluxeTransactionServerCase2, AccountDatabaseCase2)
   BankCase3_1: all (CashierTerminalCase1, CashierValidationServerCase1,
       TransactionServerManagerCase1, EconomicTransactionServerCase1)
   BankCase4_1: all (CashierTerminalCase1, CashierValidationServerCase1,
       TransactionServerManagerCase1, DeluxeTransactionServerCase1,
       AccountDatabaseCase1)
   BankCase4_2: all (CashierTerminalCase1, CashierValidationServerCase1,
       TransactionServerManagerCase1, DeluxeTransactionServerCase2,
       AccountDatabaseCase2)
2. Constraint Expression
   2.1 Default Constraint
       default (BankCase2: BankCase2_1)
       default (BankCase4: BankCase4_1)
```

4.3.3.1.3 Architecture Model Mapping (AMM)

This mapping provides the transformation for an AMDNF form from the abstract component level to the function/interface level. The transformation uses the default constraint information provided by the AMDNF at the function/interface level. Table 4.13 shows an example of the AMM from the banking domain.

Table 4.13 An Example of AMM

```
AMM for the Banking Domain Example

map (BankCase1: BankCase1_1)
map (BankCase2: BankCase2_1)
map (BankCase3: BankCase3_1)
map (BankCase4: BankCase4_1)
```

4.3.3.1.4 Abstract Component Interaction Model (ACIM)

This model describes how the abstract components interact with each other. It provides information about the initiator and responder for each component interaction. This model provides important configuration knowledge. The system generation framework depends on this knowledge to configure a concrete instance of a DCS domain. How the system generation framework uses this knowledge is described in Chapter 6. Table 4.14 shows an example of ACIM from the banking domain.

Table 4.14 An Example of ACIM

ACIM for the Banking Domain Example
interact (CashierTerminal, CashierValidationServer) interact (ATM, CustomerValiationServer) interact (CashierTerminal, TransactionServerManager) interact (CashierTerminal, EconomicTransactionServer) interact (CashierTerminal, DeluxeTransactionServer) interact (ATM, TransactionServerManager) interact (ATM, EconomicTransactionServer) interact (ATM, DeluxeTransactionServer) interact (DeluxeTransactionServer, AccountDatabase)

4.3.3.1.5 Component-Level Multiplicity Model (MM)

The multiplicity model defines the multiplicity constraints in a system. The multiplicity constraints are defined at two levels: system-level multiplicity model and component-level multiplicity model. The system-level MM is discussed in Section 4.3.2.4. The component-level multiplicity expresses the multiplicity of each pair of interaction components. This is one of the configuration knowledge used to assemble a system by the system generation framework. How the system generation framework uses this knowledge is described in Chapter 6. Table 4.15 shows an example of component-level MM from the banking domain.

Table 4.15 An Example of Component-level MM

Component-level Multiplicity Model for the Banking Domain Example
multiplicity ((CashierValidationServer, CashierTerminal) : 1..*)
multiplicity ((CustomerValiationServer, ATM) : 1..*)
multiplicity ((TransactionServerManager, CashierTerminal) : 1..*)
multiplicity ((EconomicTransactionServer, CashierTerminal) : 1..*)
multiplicity ((DeluxeTransactionServer, CashierTerminal) : 1..*)
multiplicity ((TransactionServerManager, ATM) : 1..*)
multiplicity ((EconomicTransactionServer, ATM) : 1..*)
multiplicity ((DeluxeTransactionServer, ATM) : 1..*)
multiplicity ((DeluxeTransactionServer, AccountDatabase) : 1)

4.3.3.2 Design-Feature-Related Models

Design feature related models describe functional aspects of the design features that form the architecture for a DCS family. These models include Interface Model (IM), Abstract Component Interface Model (ACIM) and Abstract Component Model (ACM).

4.3.3.2.1 Interface Model (IM)

An IM includes all the interfaces designed for a DCS family. An abstract component must implement one or more of these interfaces. An abstract component can also require one or more of these interfaces from its post-processing collaborator(s) in order to accomplish its task. Table 4.16 shows an excerpt of *IAccountDatabase* designed for the banking domain example. Complete examples can be found in both Chapter 5 and Appendix E. The collection of all the interfaces for the domain forms the IM.

4.3.3.2.2 Abstract Component Interface Model (ACIM)

An ACIM shows the required and provided interfaces of all the abstract components in a DCS family. That is, it defines the functional aspect of the abstract components. Table 4.17 shows an excerpt of ACIM from the banking domain. The complete example can be found in Section 5.3.5 or Appendix J.

Table 4.16 An Example of an Interface

```
An Interface Designed for the Banking Domain Example

Interface: IAccountDatabase
1. Syntax
Account getAccount(String accountNumber, int accountType);
    Pre: NONE
    Post: NONE
    Invariant: NONE
    Communication Pattern: cp2s or cp2a
    Description: This function returns an account object as identified by the parameters. It
        returns null if the account specified does not exist.
…
2. Variation
    IAccountDatabase: one-of (IAccountDatabaseCase1, IAccountDatabaseCase2)
    IAccountDatabaseCase1: {cp2s}
    IAccountDatabaseCase2: {cp2a}
3. Default
    default (IAccountDatabase: IAccountDatabaseCase1)
```

Table 4.17 An Example of ACIM

```
Abstract Component Interface Model for the Banking Domain Example

1. Disjunctive Normal Form
    DeluxeTransactionServer: one-of (DeluxeTransaxtionServerCase1,
        DeluxeTransactionServerCase2)
    AccountDatabase: one-of (AccountDatabaseCase1, AccountDatabaseCase2)
    EconomicTransactionServer: EconomicTransactionServerCase1
    TransactionServerManager: TransactionServerManagerCase1
    CashierTerminal: CashierTerminalCase1
    ATM: ATMCase1
    CashierValidationServer: CashierValidationServerCase1
    CustomerValidationServer: CustomerValidationServerCase1

    interface (DeluxeTransaxtionServerCase1: provided_interface  (IAccountManagementCase1,
        ICustomerManagementCase1), required_interface ( IAccountDatabaseCase1))
    interface (DeluxeTransactionServerCase2: provided_interface  (IAccountManagementCase1,
        ICustomerManagementCase1), required_interface (IAccountDatabaseCase2))
    …(continuing interface description for the rest of the abstract component listed above)

2. Constraint Expression
    2.1 Default Constraint
        default (DeluxeTransactionServer : DeluxeTransactionServerCase1)
        default (AccountDatabase : AccountDatabaseCase1)
    2.2 Satisfaction Constraint
        mutual_require (DeluxeTransactionServerCase1, AccountDatabaseCase1)
        mutual_require (DeluxeTransactionServerCase2, AccountDatabaseCase2)
```

4.3.3.2.3 Abstract Component Model (ACM)

This model consists of the UMM descriptions for all the abstract components in the domain. The detail of the UMM description including its format is discussed in Chapter 3. Examples from the banking domain can be found in Appendix F.

4.4.3.3 QoS-Related Models

The QoS related Models are the solutions in the USGPF to validate the system QoS for an assembled DCS. These models include Critical Use Case Model at function/interface level, Architecture Model in Disjunctive Normal Form and Critical Use Case Model Mapping at function/interface level, and QoS Composition and Decomposition Model. All these models help to achieve static system QoS validation by QoS composition and decomposition. Another QoS related model is the Event Grammar Model for system behavior modeling, which is for dynamic system QoS validation and is an ongoing effort.

4.4.3.3.1 Critical Use Case Model (Function/Interface Level)

In the UGDM, the critical use cases are those that are important from the angel of the system performance. Typically, the critical use cases are only a subset of the total use cases of a system. Rarely, they can be the same. Each use case consists of a set of scenarios that describe the sequence of actions required to execute the use case. Not all of the scenarios belonging to a critical use case will be important from the QoS perspective. In the critical use case model, only the most important scenario of a critical use case is considered. The Critical Use Case Model (CUCM) at the function/interface level is one of the important factors used while creating the QoS Composition and Decomposition Model (QCDM), which is discussed in Section 4.4.3.3.3. Table 4.18 shows an example of the CUCM from the banking domain.

Table 4.18 An Example of CUCM

Critical Use Case Model of the Banking Domain Example
1. Disjunctive Normal Form CriticalUseCaseModel: one-of (CriticalUseCaseModel1, CriticalUseCaseModel2, CriticalUseCaseModel3) CriticalUseCaseModel1: all (DepositMoneyCase1_1, WithdrawMoneyCase1_1, TransferMoneyCase1_1) CriticalUseCaseModel2: all (DepositMoneyCase1_2, WithdrawMoneyCase1_2, TransferMoneyCase1_2) CriticalUseCaseModel3: all (DepositMoneyCase2, WithdrawMoneyCase2, TransferMoneyCase2) DepositMoneyCase1_1: path_f(CashierTerminal.deposit[cp2s], DeluxeTransactionServer.deposit[cp2s], AccountDatabase.getAccount[cp2s], AccountDatabase.saveAccount[cp2s]) DepositMoneyCase1_2: path_f (CashierTerminal.deposit[cp2s], DeluxeTransactionServer.deposit[cp2s], AccountDatabase.getAccount[cp2a], AccountDatabase.saveAccount[cp2a]) …(continuing critical use case description for the rest of the critical use cases appear above) 2. Constraint Expression 2.1 Default Constraint default (CriticalUseCase: CriticalUseCase3)

Table 4.19 An Example of AMDNF and CUCM Mapping (Function/Interface Level)

AMDNF and CUCM Mapping (Function/Interface Level) for the Banking Domain Example
map (BankCase1_1: CriticalUseCaseModel3) map (BankCase2_1: CriticalUseCaseModel1) map (BankCase2_2: CriticalUseCaseModel2) map (BankCase3_1: CriticalUseCaseModel3) map (BankCase4_1: CriticalUseCaseModel1) map (BankCase4_2: CriticalUseCaseModel2)

4.4.3.3.2 Architecture Model in Disjunctive Normal Form and Critical Use Case Model Mapping (Function/Interface Level)

The mapping from the AMDNF to the CUCM at the function/interface level in a DCS family provides the solution to relate the system architecture to the static system QoS validation mechanism which is based on the CUCM. How to achieve this mapping and how to derive QoS composition and decomposition based on the CUCM are discussed in Chapter 5. Table 4.19 is an example of this mapping from the banking domain.

Table 4.20 An Example of QCDM

QoS Composition and Decomposition Model of the Banking Domain Example

QCDM: one-of(CriticalUseCaseModel1, CriticalUseCaseModel2, CriticalUseCaseModel3)

- CriticalUseCaseModel1

    1) QoS Composition Model
    1.1) QoS Composition Rules for throughput
    System.QoS.throughput = CriticalUseCaseModel1.QoS.throughput
    CriticalUseCaseModel1.QoS.throughput = min (DepositMoneyCase1_1.QoS.throughput, WithdrawMoneyCase1_1.QoS.throughput, TransferMoneyCase1_1.QoS.throughput)

    1/DepositMoneyCase1_1.QoS.throughput = 1/CashierTerminal.deposit.QoS.throughput + 1/DeluxeTransactionServer.deposit.QoS.throughput + 1/AccountDatabase.getAccount.QoS.throughput + 1/AccountDatabase.saveAccount.QoS.throughput
    …(continuing description of rules for throughput for the rest of the use cases shown above
    1.2) QoS Composition Rules for endToEndDelay
    …
    2) QoS Decomposition Model
    2.1) QoS Decomposition Rules for throughput
    CashierTerminal.deposit. QoS.throughput > System.QoS.throughput
    CashierTerminal.withdraw.QoS.throughput > System.QoS.throughput
    CashierTerminal.transfer.QoS.throughput > System.QoS.throughput
    DeluxeTransactionServer.deposit. QoS.throughput > System.QoS.throughput
    DeluxeTransactionServer.withdraw.QoS.throughput > System.QoS.throughput
    DeluxeTransactionServer.transfer.QoS.throughput > System.QoS.throughput
    AccountDatabase.getAccount. QoS.throughput > System.QoS.throughput
    AccountDatabase.saveAccount.QoS.throughput > System.QoS.throughput
    2.2) QoS Decomposition Rules for endToEndDelay
    …

- CriticalUseCaseModel2

    …

- CriticalUseCaseModel3

    …

4.4.3.3.3 QoS Composition and Decomposition Model (QCDM)

This model describes the QoS composition and decomposition rules for each required QoS parameter for each critical use case. These rules form the QoS Composition and Decomposition Model (QCDM). The QCDM for each critical use case consists of two models: QoS Composition Model and QoS Decomposition Model. The statements in the QoS Composition Model are arranged in a “hierarchical form”, i.e., the first statement expresses the formula for calculating the value for a QoS parameter. The rest of the statements express how to calculate the values of variables in the right hand side of the first statement. The QoS Decomposition Model consists of statements for deriving QoS parameters for each function call for a component involved in the critical use cases. The format of this section is a listing of QCDM for each critical use case in the Critical Use Case Model. Table 4.20 is an excerpt of QCDM from the banking domain which also shows the template for documenting this model. The complete example can be found in Appendix G and the description can be found in Section 5.3.10.

This chapter describes the contents of the UGDM and the UDSL for modeling and describing a UGDM in the USGPF. The UGDM is important for representing a DCS family, including the QoS aspect. As the UGDM becomes more comprehensive, more models may be included, for example, Event Grammar Model [AUG95, AUG97] for describing the system behavior. Next chapter presents the UniFrame UGDM Development Process (UGDP) which is a process for creating such a UGDM for a distributed computing domain.

5 The UNIFRAME UGDM DEVELOPMENT PROCESS (UGDP)

Chapter 4 gives the detailed description on the UGDM. This chapter provides the UniFrame UGDM Development Process (UGDP), which is for creating the UGDM for a selected domain. The UGDP is the second part of the UniFrame System-Level Generative Programming Framework (USGPF). The UGDP covers the generative domain engineering of the UA. It is a use-case-driven, architecture-centric and iterative process. Of critical importance is that the UGDP must be domain-independent; repeatability across multiple domains is an essential requirement. A banking domain example is completely developed throughout this chapter to demonstrate the UGDP and is also used to test validity of the proposed USGPF.

5.1 Overview of the UGDP

The outline of the UGDP is shown in Table 5.1. The UGDP consists of three phases: *Domain Analysis*, *Domain Design* and *Ordering DSL Design*. The *Domain Analysis* phase involves *Domain Definition* and *Domain Modeling*. This phase is similar to the *Domain Analysis* in DEMRAL. The purpose of *Domain Definition* is to establish the domain scope based on the analysis of stakeholders, their goals and existing systems. The purpose of *Domain Modeling* is to model the contents of the domain by finding the relevant domain concepts and modeling their features. In *Domain Modeling*, both the functional and QoS requirements are identified. In the *Domain Design* phase the common layered architecture for a DCS family is developed as well as various QoS related models. In the phase of *Ordering DSL Design*, ordering schemes are designed so that application engineers or system assemblers can order a DCS by supply system requirements. This ordering language can be textual, tabular, or graphic. It can also be supported by natural language processing.

Table 5.1 Outline of the UGDP

The UniFrame UGDM Development Process (UGDP)
1. Domain Analysis
1.1 Domain Definition
1.1.1 Domain Description
1.1.2 Domain Scoping and Context Analysis
1.2 Domain Modeling
1.2.1 Modeling Domain Functional Requirements
1.2.2 Identifying and Modeling Domain Key Concepts
1.2.3 Identifying and Modeling Domain QoS Requirements
2. Domain Design
2.1 Designing Layered Architecture
2.2 Creating Component Diagrams
2.3 Creating Sequence Diagrams
2.4 Refining Critical Use Case Model to Abstract Component Level
2.5 Identifying Component Interfaces and Communication Patterns
2.6 Refining Critical Use Case Model to Function/Interface Level
2.7 Refining Architecture Model in Disjunctive Normal Form from Component Level to Function/Interface Level
2.8 Mapping Architecture Model in Disjunctive Normal Form to Critical Use Case Model (Function/Interface Level)
2.9 Creating Abstract Component Model
2.10 Creating QoS Composition and Decomposition Model
3. Ordering Language Design

The details of the process are described in next sections. Each step in the process is further illustrated through a banking domain example. The outcome of this example is the UGDM presented Appendix I.

5.2 Domain Analysis

Domain in the UniFrame Approach refers to industry specific markets such as Financial Services, Health Care Services and Manufacturing Services. Domain analysis involves two main activities: *Domain Definition* and *Domain Modeling*. The purpose of *Domain Definition* is to establish the domain scope based on the analysis of stakeholders, their goals, and existing systems. The purpose of *Domain Modeling* is to model the contents of the domain by finding the relevant domain concepts and modeling their features. At the beginning of *Domain Analysis*, establish a domain dictionary and a register of domain knowledge sources. Domain dictionary includes definitions of domain features and concepts. Domain knowledge sources are references to the literature, manuals, and domain experts consulted during domain analysis. This information is updated as the process going on.

5.2.1 Domain Definition

Domain Definition involves *Domain Description*, and *Domain Scoping and Context Analysis*. The first activity of *Domain Definition* is to identify stakeholders and their goals. The next activity is to determine the scope and characterize the contents of the domain.

5.2.1.1 Domain Description

This step follows Varghese's work on the problem space of a variable domain [VAR02]. The goal of this step is to obtain an initial understanding of the domain which is going to be modeled. This is important because it gives everyone involved an initial understanding of what is going to be accomplished. This should begin with the development of a problem statement. Although this may not be very detailed and well defined at the beginning, it is important to document the overall goal at the beginning of the domain engineering process. The next item to be produced is a general description of the capabilities that applications falling within this domain should possess. This should include any desired properties of the system family that have not yet been captured in the problem statement. The final item to be produced is a list of any existing applications that would fall under the description of this domain.

Table 5.2 Domain Description for the Banking Domain Example

Domain Description
1. Problem Description To create a banking system that is able to manage account activities. 2. Description of General Capabilities The system should be able to process the basic account functions: create an account, delete an account, query account balance, deposit money and withdraw money. The system may contain different security features, including client security and server security. The system must meet the QoS requirements. The domain includes interfaces for bank staff to manage accounts and may also include interfaces (i.e. ATM) for customers to manage their accounts. 3. Domain Boundaries This is a simple banking domain (version 1.0) to provide the basic personal account management. Corporate account management is not considered. Advanced banking features, for example, loan processing and credit card, are not supported. 4. Potential Sources of Information 1. Banking Staff – They have knowledge regarding required features and rules. 2. Application Engineers – May have applicable knowledge from past developments of related applications. May also have knowledge regarding system requirements at various sites. 3. Literature – May have formal definitions of key terms, example models, etc. 5. Potential Stakeholders and Experts 1. Senior Management 2. Project Leaders 3. Application Engineers 4. Bank Staffs 6. Related Domain 1. Loan Domain

The stakeholder analysis is a dynamic, social process, which may involve not only identifying the key players for a domain, but also getting some important people or organizations to be involved. These people may have oversight responsibilities or they may be a resource for better understanding of the domain. For the banking domain example, four groups of people were recognized as important stakeholders. Senior management and project leadership were included to ensure that they remain aware of ongoing progress. Application engineers are important as a source of information about any previous projects that may be related to the current project. They will also be the key people later developing the software solution. Bank staffs are the end users of this system. The study of the resources and meeting with the stakeholders will expand the domain description, including defining the domain boundaries. The domain boundaries state clearly what the application is going to be. A list of potential sources of information is also identified. Any relationships to other domains that can be identified also need to be documented. This is also a source for gaining insight into the current domain. The banking domain has a lot of similarities to the loan domain. For example, in the banking domain, customers have their accounts, they save money and get interest from a bank; in the loan domain, customers also have their accounts; however, they borrow money, and they pay loans and interests. The studying of the loan domain can provide useful input to the banking domain. Table 5.2 shows the artifact of domain description for the banking domain example.

5.2.1.2 Domain Scoping and Context Analysis

This is to determine the scope and characterize the contents of the domain by defining its domain features. The domain features are obtained by analyzing the application areas and markets of the systems in the domain and by analyzing the existing systems.

By analyzing the domain description we derived, studying exemplar systems, consulting domain experts, a use case model (UCM) is developed to formally define the domain functionalities. Figure 5.1 demonstrates the use case model of the banking domain example. Direct users of the system include cashiers and customers. An account can only be accessed by one user at a time. Table 5.3 shows the description of the use case model for the banking domain example. Table 5.4 shows an example of a domain dictionary for the banking domain example. The table is a partial listing. The description of the use case model is also a part of the domain dictionary.

Use cases are most often described from an end-user point of view. For example, with an automated teller machine (ATM), we might investigate use cases for the customers such as *DepositMoney*, *WithdrawMoney*, *TransferMoney*, and *CheckBalance*, etc.

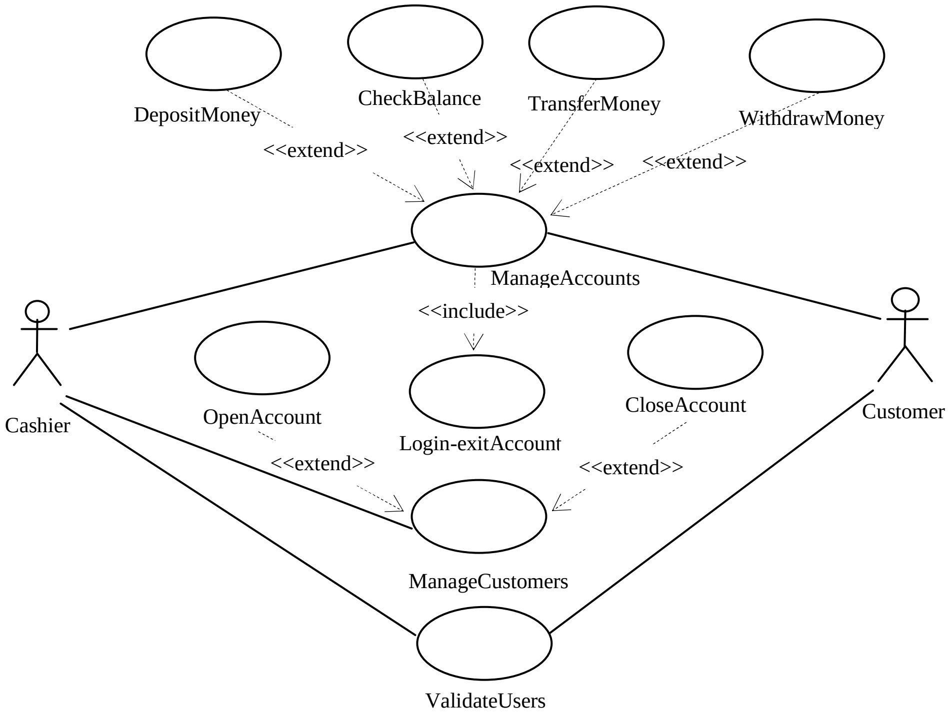


Figure 5.1 UCM for the Banking Domain Example

Table 5.3 Description of the UCM for the Banking Domain Example

Description of Use Case Model
ManageAccounts: Common account activities, including depositing money, withdrawing money, transferring money, and checking balance. Both cashiers and customers have activities of ManageAccounts. DepositMoney: An activity of ManageAccounts. Depositing certain amount of money into an account. WithdrawMoney: An activity of ManageAccounts. Withdrawing certain amount of money from an account. TransferMoney: An activity of ManageAccounts. Transferring certain amount of money from one account to another. CheckBalance: An activity of ManageAccounts. Checking the balance of an account. ManageCustomers: Activities of opening an account or closing an account for a customer. ManageCustomers is intended for cashiers only. OpenAccount: An activity of ManageCustomers. Opening an account for a customer. CloseAccount: An activity of ManageCustomers. Closing an account for a customer. Login-exitAccount: An activity used by activities of ManageAccounts. The login process checks whether the specified account exists, if it exists, locks the account so that other activities can not access the account in order to ensure data integrity. The exit process unlocks an account so that other activities can use the account. ValidateUsers: Validating cashiers and customers before they can use a bank system. It is a password checking process. The user name for a cashier is his/her user id. The user name for a customer is his/her account number.

Table 5.4 Domain Dictionary for the Banking Domain Example

Domain Dictionary (partial listing)
Banking: domain for managing personal accounts. Bank: An entity keeping accounts. Cashier: Persons who manage personal accounts on behalf of customers. Customer: Persons who owns accounts. User: Cashier and Customer. Account: An entity keeping the money belongs to a customer …

5.2.2 Domain Modeling

Domain Modeling involves three activities: modeling the domain functional requirements, identifying and modeling the domain key concepts, and identifying and modeling the domain QoS requirements.

5.2.2.1 Modeling Domain Functional Requirements

The use case model (UCM) established above represents the functional requirements of a domain. However, the use case diagram cannot express the common and variable functional properties which are the inherent characteristics of a DCS family. For example, in the banking domain example, both cashiers and customers manage accounts; however, a system might not provide this functionality for customers. The variability can be easily modeled in a feature diagram. A feature diagram is a concise and convenient way of defining a domain. It is used throughout the UGDP to document the common and variable properties of different artifacts. The feature diagram shown in Figure 5.2 captures the common and variable properties of the functional requirements for the banking domain example, which is also expressed in the UDSL as shown in Table 5.5 with all the constraints that can not be expressed by the feature diagram.

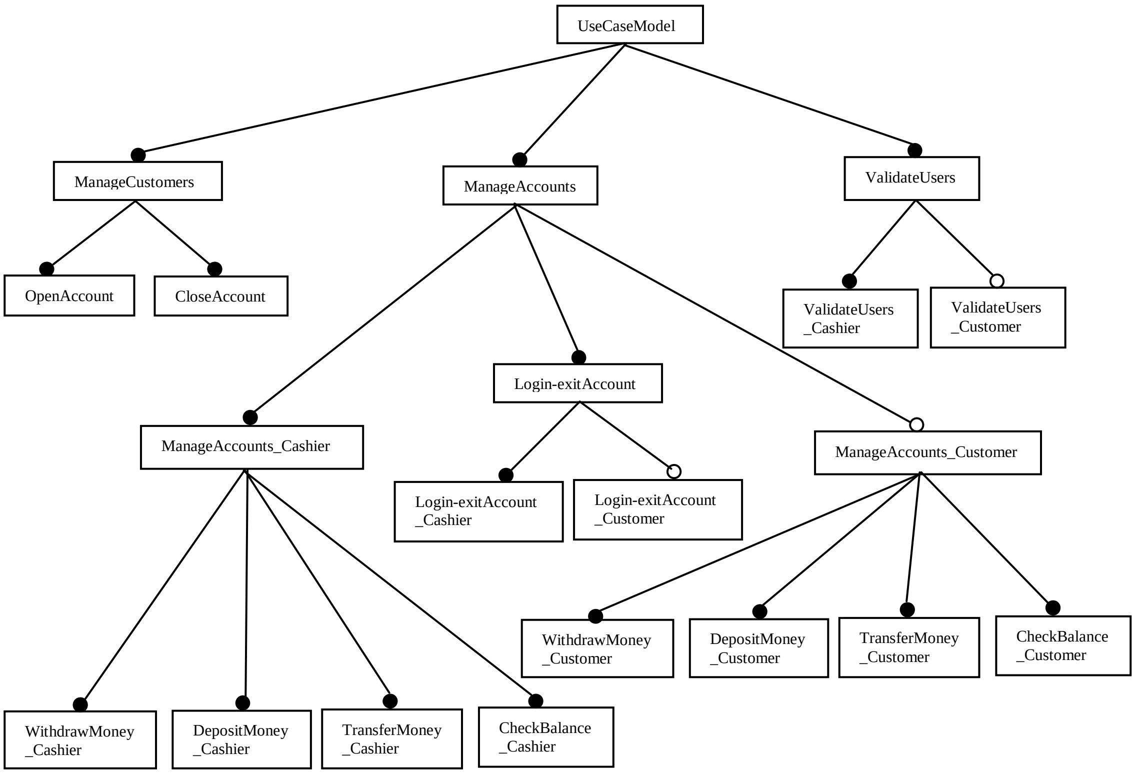


Figure 5.2 Feature Diagram of the UCM for the Banking Domain

Table 5.5 UCM in the UDSL for the Banking Domain Example

Use Case Model
1. Commonality and Variation Bank: all (ManageCustomers, ManageAccounts, Login-exitAccount, ValidateUsers) ManageCustomers: all (OpenAccount, CloseAccount) ManageAccounts: all (ManageAccounts_Cashier, ManageAccounts_Customer?) ManageAccounts_Cashier: all (WithdrawMoney_Cashier, DepositMoney_Cashier, TransferMoney_Cashier, CheckBalance_Cashier) ManageAccounts_Customer: all (WithdrawMoney_Customer, DepositMoney_Customer, TransferMoney_Customer, CheckBalance_Customer) ValidateUsers: all (ValidateUsers_Cashier, ValidateUsers_Customer?) Login-exitAccount: all (Login-exitAccount_Cashier, Login-exitAccount_Customer?) 2. Constraint Expression 2.1 Default Constraint default (ManageAccounts : ManageAccounts_Cashier) default (ValidateUsers : ValidateUsers_Cashier) default (Login-exitAccount : Login-exitAccount_Cashier) 2.2 Satisfaction Constraint mutual_require (ValidateUsers_Customer, ManageAccounts_Customer, Login-exitAccount_Customer)

5.2.2.2 Identifying and Modeling Domain Key Concepts

Source of key concepts and features includes existing and potential stakeholders, domain experts and domain literature, existing systems, preexisting models (e.g., use case models, object models), etc. Strategies for identifying features include both top-down approaches and bottom up approaches. In the banking domain example, *Account* is identified as a key concept. An *Account* has many common features, including *Account Number*, *Customer Name*, *Balance* and *Account Type*. It may also have an *Interest Rate*, depending on what type of *Account* it is. Figure 5.3 and Table 5.6 show the modeling of the concept of *Account* that is used in the banking domain example.

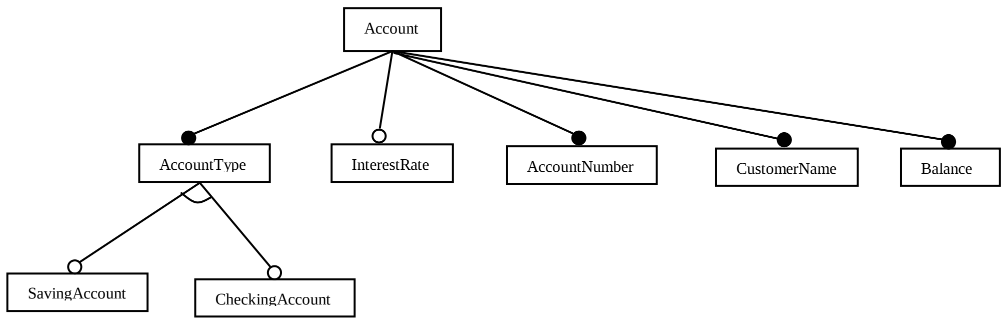


Figure 5.3 Feature Diagram of Key Concepts for the Banking Domain Example

Table 5.6 Key Concepts in the UDSL for the Banking Domain Example

Key Concepts in the UDSL

1. Commonality and Variation
    Account: all (AccountType, InterestRate?, AccountNubmer, CustomerName, Balance)
    AccountType: one-of (SavingAccount, CheckingAccount)
 	
2. Constraint Expression
    2.1 Satisfaction Constraint
        require (SavingAccount, InterestRate)
        reject (CheckingAccount, InterestRate)

5.2.2.3 Identifying and Modeling Domain QoS Requirements

QoS is an inherent characteristic of the UniFrame. It is important to identify and model the domain QoS requirements in order to build QoS-aware DCS. There are two steps to do so. Firstly, consult the QoS catalog and domain experts to identify the key QoS parameters for evaluating or monitoring the system. Secondly, identify the critical use cases in the system. The critical use cases are a subset of the use cases identified for a domain. The evaluation and monitor of the QoS parameters on these critical use cases can represent the QoS parameters of the system. The use of critical use case in QoS evaluation and monitoring can make the process simpler and more effective. The outcome of this step is two models: the QoS requirement model (QRM) and the critical use case model (CUCM).

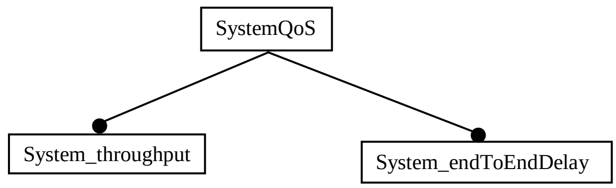


Figure 5.4 QRM for the Banking Domain Example

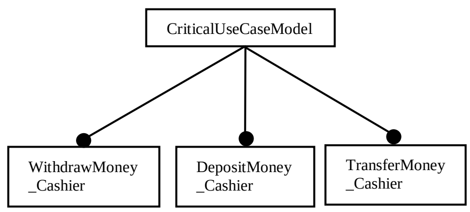


Figure 5.5 CUCM for the Banking Domain Example

Suppose the analysis of the QoS criteria reveals that *throughput* and *endToEndDelay* are the two critical QoS features that are needed in banking systems and are the two standard measurements for the system performance. Suppose *DepositMoney*, *WithdrawMoney* and *TransferMoney* of cashiers are the critical use cases of bank systems. There are four ways to represent the system QoS from the critical use case QoS: minimal QoS of the critical use cases, maximal QoS of the critical use cases, a customized expression (one special case is taking the average), or providing QoS of all the critical use cases. For the first three ways, each system QoS parameter is expressed as one value. For the last one, each system QoS parameter is expressed as a set of values on different critical use cases. In the banking domain example, the minimal QoS value is adopted for *throughput*, and the maximal QoS value is adopted for *endToEndDelay*. Figure 5.4 and Figure 5.5 show the QoS requirement model and the critical use case model for the banking domain example respectively. Table 5.7 and Table 5.8 show these models in the UDSL respectively.

Table 5.7 QRM in the UDSL for the Banking Domain Example

QoS Requirement Model (QRM) in the UDSL
SystemQoS: all (System_throughput, System_endToEndDelay)

Table 5.8 CUCM in the UDSL for the Banking Domain Example

Critical Use Case Model (CUCM) in the UDSL
CriticalUseCaseModel: all (WithdrawMoney_Cashier, DepositMoney_Cashier, TransferMoney_Cashier)

5.3 Domain Design

The goal of *Domain Design* is to develop the layered architecture for a DCS family as well as various QoS related models. The architectural view of a design model presents the most architecturally important classifiers of the design model: the most important subsystems, interfaces, as well as a few very important classes, primarily the active classes. It also presents how the most important use cases are realized in terms of these classifiers. There are different definitions of software architecture. Following are the two popular definitions.

Shaw and Garlan [SHA96] define software architecture as follows. Abstractly, software architecture involves the description of elements from which systems are built, interactions among those elements, patterns that guide their composition, and constraints on these patterns. In general, a particular system is defined in terms of a collection of components and interactions among these components. Such a system may in turn be used as a (composite) element in a larger system design.

Buschmann et al. [BUS96] offer another definition of software architecture. A software architecture is a description of the subsystems and components of a software system and the relationship among them. Subsystems and components are typically specified in different views to show the relevant functional and nonfunctional properties of a software system. The software architecture of a system is an artifact. It is the result of the software development activity.

The architecture developed in the UGDP follows both definitions above. The architectural design of a system is a high-level design. The goal is to come up with a flexible structure which supports structural variation in its topology. This kind of architecture satisfies all important requirements and still leaves a large degree of freedom for the implementation. As a design rule, use the most stable parts of a DCS family to form the “skeleton” and make the rest flexible and easy to evolve and maintain. But even the skeleton has to be modified sometimes, especially when a UGDM has not reached its maturity.

5.3.1 Designing Common Layered Architecture

The development of a common architecture for a family of systems is a critical step. This architecture indicates the commonality and variability. Designing the architecture is an iterative process. It requires analyzing the requirement model and the design of existing systems and meeting with persons who have built many systems for different customers in the same problem area. It usually needs prototyping.

Buschmann et al. summarized a list of architectural patterns in [BUS96]: layers pattern, pipes and filters pattern, blackboard pattern, broker pattern, model-view-controller pattern, and microkernel pattern. The advantage of the layers pattern is the modularization of a system. When a layer is modified, it has the minimal impact on the overall system structure. This makes the refinement and maintenance of an architecture easier and less error prone.

In the UniFrame, a layering pattern is adopted for the system architecture. This layering is achieved by decomposing tasks into groups of subtasks, in which each group of subtasks is at a particular level of abstraction.

The process of designing a common layered architecture for a family of systems involves answering questions such as what kinds of subsystems and/or abstract components are needed to meet certain functional or nonfunctional requirements, how these subsystem and/or abstract components are connected, what are the constraints, what kind of middleware or component model will be used, what interfaces the abstract components will have, how they will accommodate the requirements, etc. The process typically begins by looking at a few use cases, creating use case realizations for them, and identifying the roles for the design features. Then do the same for other use cases. As the work continues we should be able to identify the design features and design variations that are needed for designing a common layered architecture.

There are three categories of design features: *system*, *subsystem* and *abstract component*. The top root of the feature diagram of a layered architecture is a *system* design feature, which is denoted by its name surrounded by <>. The leaves of a design feature diagram are *abstract component*s, which are denoted by their names surrounded by []. The other nodes in the design feature diagrams are *subsystem*s, which are denoted by their names surrounded by (). The *subsystem*s are the aggregation of *subsystem*s and/or *abstract component*s. The *abstract component*s are atomic features and the *subsystem*s are composite features. The *abstract component*s will be realized in the component engineering stage.

For the banking domain example, suppose the analysis shows the need for a user subsystem to accept requests from users (cashiers and customers), a transaction subsystem for carrying out account and customer management, and a user validation subsystem. The user subsystem passes the requests to the user validation subsystem and the transaction subsystem. The account and customer management use cases are realized by the user subsystem and the transaction subsystem. The user validation use case is realized by the user subsystem and the user validation subsystem. The user validation subsystem should be able to validate both cashiers and customers of a bank. The design at this first layer is documented as Architecture Model in Hierarchical Form (AMHM) in Figure 5.6, Design Feature Interaction Model (DFIM) in Figure 5.7, constraints in Table 5.9 and design feature description in Table 5.10. In the DFIM, the notation "I" indicates the design feature that initiates the interaction between the two associated design features.

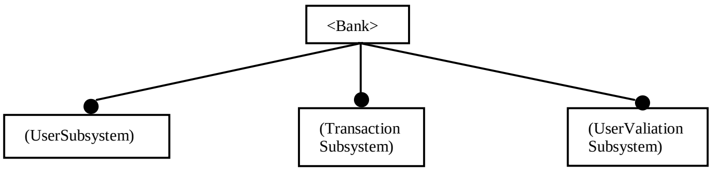


Figure 5.6 Feature Diagram of AMHF for the Banking Domain Example (Layer 1)

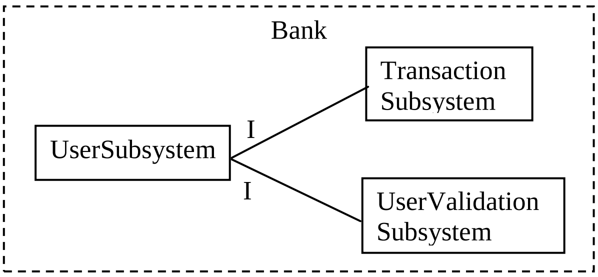


Figure 5.7 DFIM for the Banking Domain Example (Layer 1)

Table 5.9 Constraints in the UDSL for the Banking Domain Example (Layer 1)

Constraints in the UDSL
1. Multiplicity Constraint multiplicity ((Bank, UserSubsystem) : 1) multiplicity ((Bank, TransactionSubsystem) : 1) multiplicity ((Bank, UserValidationSubsystem) : 1) 2. Default Constraint NONE 3. Satisfaction Constraint NONE

Table 5.10 Design Feature Description for the Banking Domain Example (Layer 1)

Design Feature Description
1. System Bank: Provide basic account management and transaction services. 2. SubSystem UserSubsystem: Interact with users. UserValidationSubsystem: Validate a user before the user can use the system. TransactionSubsystem: Perform transactions. 3. Abstract Component NONE

In order to meet the possible different levels of the QoS requirements and the financial affordability of different bank corporations, multiple transaction subsystems are designed. Suppose an economic transaction subsystem with a single server and a deluxe transaction subsystem with multiple servers with dedicated functionalities are designed. For the user subsystem, customer and cashier have different need for using the system. A customer only needs to manage his/her accounts. A cashier needs not only to manage customers' accounts, but also to manage customers. For the user validation subsystem, separate abstract components are designed to validate cashiers and customers respectively. The outcome of this second layer is shown in Figure 5.8, Figure 5.9, Table 5.11 and Table 5.12.

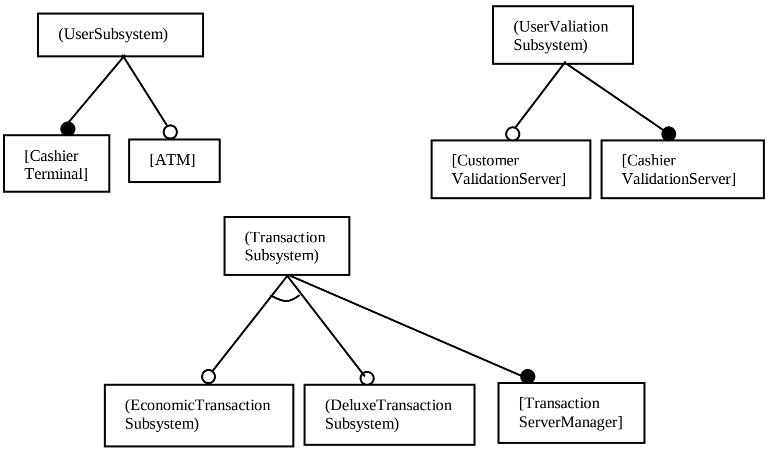


Figure 5.8 Feature Diagram of AMHF for the Banking Domain Example (Layer 2)

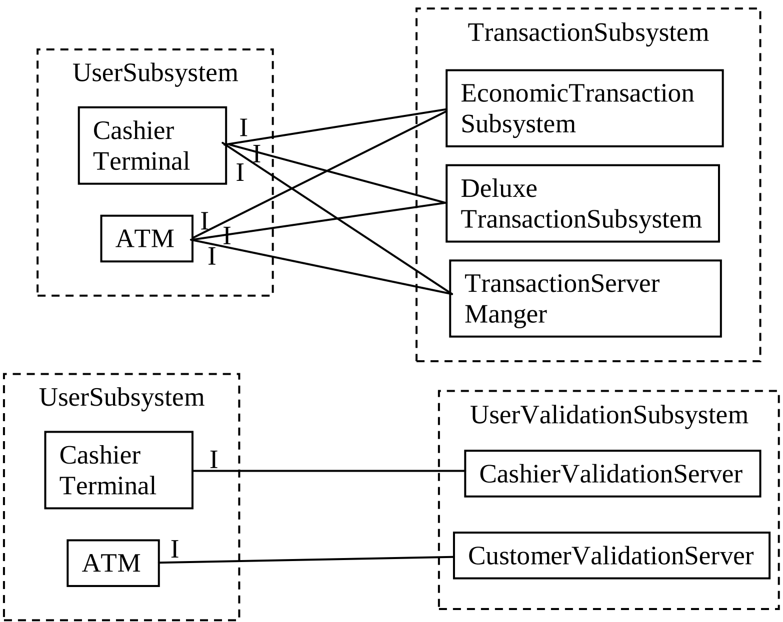


Figure 5.9 DFIM for the Banking Domain Example (Layer 2)

Table 5.11 Constraints in the UDSL for the Banking Domain Example (Layer 2)

Constraints in the UDSL
1. Multiplicity Constraint multiplicity ((UserSubsystem, CashierTerminal) : 1..*)) multiplicity ((UserSubsystem, ATM) : 0..*)) multiplicity ((UserValidationSubsystem, CustomerValidationServer): 0..1)) multiplicity ((UserValidationSubsystem, CashierValidationServer) : 1)) multiplicity ((TransactionSubsystem, EconomicTransactionSubsystem) : 1..2)) multiplicity ((TransactionSubsystem, DeluxeTransactionSubsystem) : 1..2)) multiplicity ((TransactionSubsystem, TransactionServerManager) : 1))
2. Default Constraint default (UserSubsystem : CashierTerminal) default (UserValidationSubsystem : CashierValidationServer) default (TransactionSubsystem : all (TransactionServerManager, EconomicTransactionSubsystem))
3. Satisfaction Constraint mutual_require (ATM, CustomerValiationServer)

Table 5.12 Design Feature Description for the Banking Domain Example (Layer 2)

Design Feature Description
1. System NONE
2. SubSystem EconomicTransactionSubsystem: provide account transaction service with low performance. DeluxeTransactionSubssytem: provide account transaction service with high performance.
3. Abstract Component CashierTerminal: interact with cashiers. Provide both account management and account transaction service. ATM: interact with customers. Provide only account transaction service. TransactionServerManager: Keep a list of account numbers and servers on which the accounts are stored. CustomerValidationServer: provide customer validation service for ATM. CashierValidationServer: provide cashier validation service for CashierTerminal.

The third layer of the banking domain example is the design for the economic transaction subsystem and the deluxe transaction subsystem. The outcome is shown in Figure 5.10, Figure 5.11, Table 5.13 and Table 5.14. When all the leaves in the layered architecture are abstract components, the design reaches the bottom.

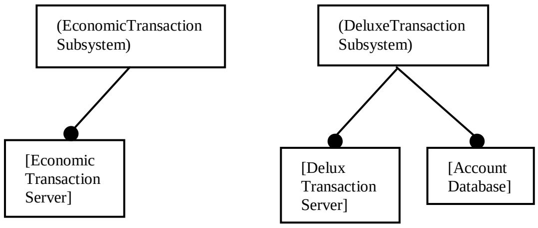


Figure 5.10 Feature Diagram of AMHF for the Banking Domain Example (Layer 3)

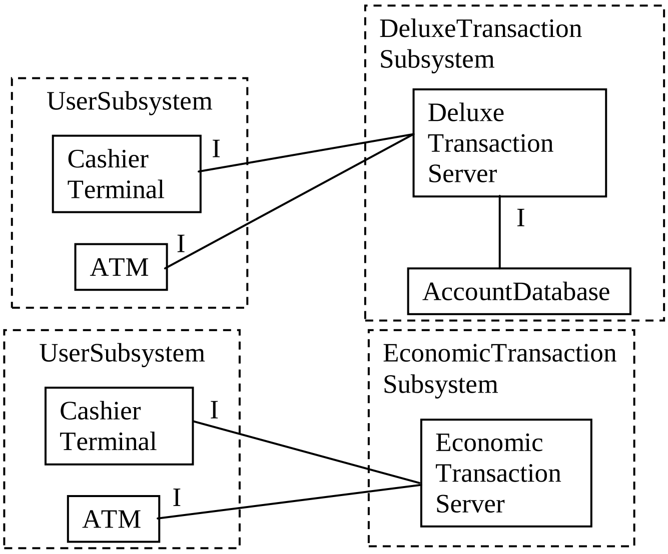


Figure 5.11 DFIM for the Banking Domain Example (Layer 3)

Table 5.13 Constraints in the UDSL for the Banking Example (Layer 3)

Constraints in the UDSL
1. Multiplicity Constraint multiplicity ((EconomicTransactionSubsystem, EconomicTransactionServer) : 1) multiplicity ((DeluxeTransactionSubsystem, DeluxeTransactionServer) : 1) multiplicity ((DeluxeTransactionSubsystem, AccountDatabase) : 1) 2. Default Constraint NONE 3. Satisfaction Constraint NONE

Table 5.14 Design Feature Description
for the Banking Domain Example (Layer 3)

Design Feature Description
1. System NONE 2. SubSystem NONE 3. Abstract Component EconomicTransactionServer: provide account transaction service with low performance. DeluxeTransactionServer: provide account transaction service with high performance. AccountDatabase: provide account storage.

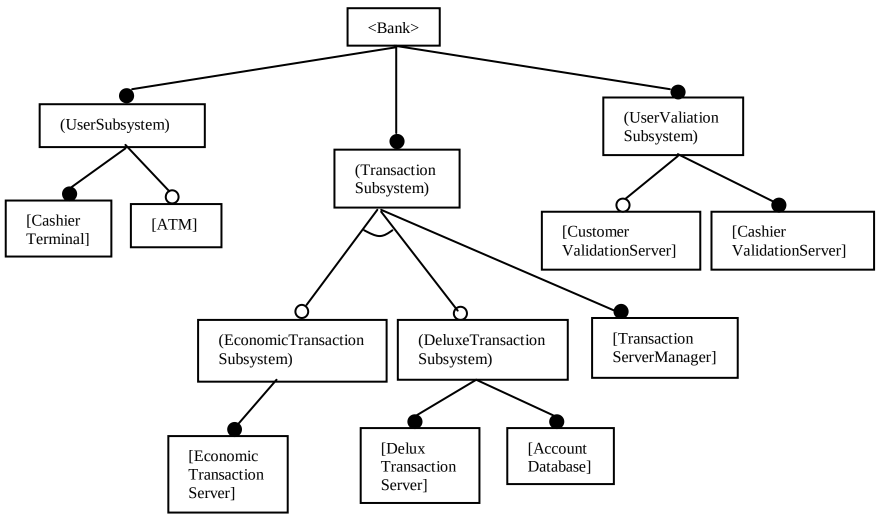


Figure 5.12 Feature Diagram of AMHF for the Banking Domain Example

Put all the increments from each layer together to derive the feature diagram of Architecture Model in Hierarchical Form (AMHF) for the banking domain example as shown in Figure 5.12, constraint as shown in Table 5.15, Design Feature Interaction Model (DFIM) in Figure 5.13, and design feature description in Table 5.16. From the banking domain example, we can see that one concept in the requirement model can be mapped to one abstract component, or mapped to a set of abstract components that form a subsystem.

Table 5.15 Constraints in the UDSL for the Banking Example

Constraints
1. Multiplicity Constraint multiplicity ((Bank,UserSubsystem): 1) multiplicity ((Bank, TransactionSubsystem) : 1) multiplicity ((Bank, UserValidationSubsystem) : 1) multiplicity ((UserSubsystem, CashierTerminal) :1..*) multiplicity ((UserSubsystem, ATM) : 0..*) multiplicity ((UserValidationSubsystem, CustomerValidationServer): 0..1) multiplicity ((UserValidationSubsystem, CashierValidationServer) : 1) multiplicity ((TransactionSubsystem, EconomicTransactionSubsystem) : 1..2) multiplicity ((TransactionSubsystem, DeluxeTransactionSubsystem) : (1..2) multiplicity ((T ransactionSubsystem, TransactionServerManager) : 1) multiplicity ((EconomicTransactionSubsystem, EconomicTransactionServer) : 1) multiplicity ((DeluxeTransactionSubsystem, DeluxeTransactionServer) : 1) multiplicity ((DeluxeTransactionSubsystem, AccountDatabase) : 1)
2. Default Constraint default (UserSubsystem : CashierTerminal) default (UserValidationSubsystem : CashierValidationServer) default (TransactionSubsystem : all (TransactionServerManager, EconomicTransactionSubsystem))
3. Satisfaction Constraint mutual_require (ATM, CustomerValiationServer)

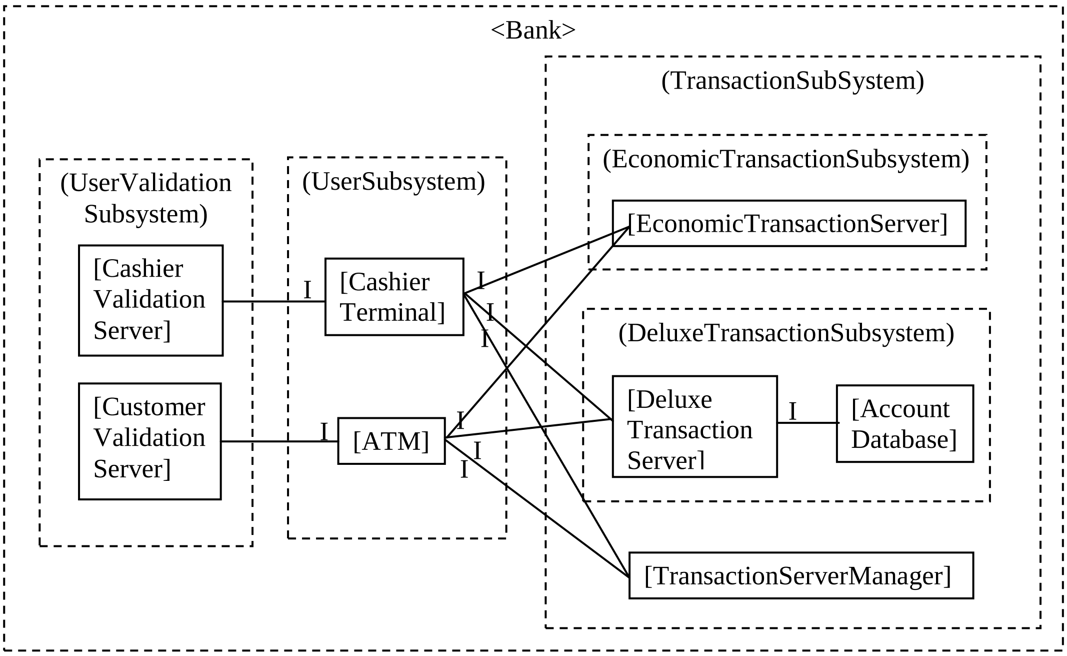


Figure 5.13 DFIM for the Banking Domain Example

Table 5.16 Design Feature Description for the Banking Example

Design Feature Description
1. System Bank: Provide basic account management and transaction services. 2. SubSystem UserSubsystem: Interact with users. UserValidationSubsystem: Validate a user before the user can use the system. TransactionSubsystem: Perform transactions. EcnomicTransactionSubsystem: provide account transaction service with low performance. DeluxeTransactionSubssytem: provide account transaction service with high performance. 3. Abstract Component CashierTerminal: interact with cashiers. Provide both account management and account transaction service. ATM: interact with customers. Provide only account transaction service. ServerManager: Keep a list of account numbers and servers on which the accounts are stored. CustomerValidationServer: provide customer validation service for ATM. CashierValidationServer: provide cashier validation service for CashierTerminal. EconomicTransactionServer: provide account transaction service with low performance. DeluxeTransactionServer: provide account transaction service with high performance. AccountDatabase: provide account storage for DeluxeTransactionServer. CustomerValidationServer: provide customer validation service for ATM. CashierValidationServer: provide cashier validation service for CashierTerminal.

Table 5.17 AMHF in the UDSL for the Banking Domain Example

Architecture Model in Hierarchical Form
Bank: all (UserSubsystem, UserValidationSubsystem, TransactionSubsystem) UserSubsystem: all (ATM?, CashierTerminal) UserValidationSubsystem: all (CustomerValidationServer?, CashierValidationServer) TransactionSubsystem: all (TransactionServerManager, one-of (EconomicTransactionSubsystem, DeluxeTransactionSubsystem)) EconomicTransactionSubsystem: EconomicTransactionServer DeluxeTransactionSubsystem: all (DeluxeTransactionServer, AccountDatabase)

Table 5.18 ACIM in the UDSL for the Banking Domain Example

Abstract Component Interaction Model
interact (CashierTerminal, CashierValidationServer) interact (ATM, CustomerValiationServer) interact (CashierTerminal, TransactionServerManager) interact (CashierTerminal, EconomicTransactionServer) interact (CashierTerminal, DeluxeTransactionServer) interact (ATM, TransactionServerManager) interact (ATM, EconomicTransactionServer) interact (ATM, DeluxeTransactionServer) interact (DeluxeTransactionServer, AccountDatabase)

Table 5.19 MMSL in the UDSL for the Banking Domain Example

System-Level Multiplicity Model
multiplicity ((Bank, CashierTerminal): 1..*) multiplicity ((Bank, ATM) : 0..*) multiplicity ((Bank, CashierValidationServer) : 1) multiplicity ((Bank, CustomerValidationServer) : 0..1) multiplicity ((Bank, TransactionServerManager) : 1) multiplicity ((Bank, EconomicTransactionServer) : 0..2) multiplicity ((Bank, DeluxeTransactionServer) : 0..2) multiplicity ((Bank, AccountDatabase) : 0..2)

Table 5.20 MMCL in the UDSL for the Banking Example

Component-level Multiplicity Model
multiplicity ((CashierValidationServer, CashierTerminal) : 1..*) multiplicity ((CustomerValiationServer, ATM) : 1..*) multiplicity ((TransactionServerManager, CashierTerminal) : 1..*) multiplicity ((EconomicTransactionServer, CashierTerminal) : 1..*) multiplicity ((DeluxeTransactionServer, CashierTerminal) : 1..*) multiplicity ((TransactionServerManager, ATM) : 1..*) multiplicity ((EconomicTransactionServer, ATM) : 1..*) multiplicity ((DeluxeTransactionServer, ATM) : 1..*) multiplicity ((DeluxeTransactionServer, AccountDatabase) : 1)

The feature diagram of the Architecture Model in Hierarchical Form shown in Figure 5.12 can be expressed in the UDSL as shown in Table 5.17. The abstract component interaction model (ACIM) can be derived from Figure 5.13. The ACIM consists of only abstract components. The ACIM in the UDSL for the banking domain example is shown in Table 5.18.

From the multiplicity constraints and the abstract component interaction model, derive the System-Level Multiplicity Model (MMSL) and the Component-Level Multiplicity Model (MMCL). The MMSL expresses the multiplicity of the root feature (a system) in terms of the leaves (abstract components). The MMCL expresses the multiplicity of each pair of interaction components. The method for deriving these two artifacts is a series of substitutions using the multiplicity constraints and the abstract component interaction model. Table 5.19 and 5.20 show these two artifacts for the banking domain example.

5.3.2 Creating Component Diagrams

From the AMHL, a normalized architecture model, i.e., Architecture Model in Normalized Form (AMNF), which consists of only abstract components can be derived. The AMNF for the banking domain example is shown in Table 5.21. The AMNF can then be transformed into disjunctive normal form, i.e., architecture model in disjunctive normal form (AMDNF). The AMDNF for the banking domain example is shown in Table 5.22. Each disjunctive normal form at the abstract component level represents one possible architecture instance. When looking at the communication pattern level, each disjunctive normal form at abstract component level may represent multiple system instances as revealed later in the process. The satisfaction constraints are used in the transformation process of architectures. Details of how to do the transformations between different forms of architecture model is discussed in Section 4.2.3.

For each disjunctive normal form at the abstract component level, there is a component diagram, which shows a set of components and their relationships. Component diagrams are used to illustrate the static implementation view of a system architecture. Component diagrams can be derived intuitively from the design feature interaction model and the component-level multiplicity model. Figure 5.14 shows the component diagram for *BankCase1*, one case in the AMNF for the banking domain example. A complete list of all component diagrams is in Appendix B.

Table 5.21 AMNF in the UDSL for the Banking Domain Example

```
                         Architecture Model in Normalized Form

1. Commonality and Variation
      Bank: all (all (ATM?, CashierTerminal), all (CustomerValidationServer?,
          CashierValidationServer), all (TransactionServerManager, one-of
          (EconomicTransactionServer, all (DeluxeTransactionServer, AccountDatabase))))

2. Constraint Expression
    2.1 Default Constraint
          default (Bank : all (CashierTerminal, CashierValidationServer,
              TransactionServerManager, EconomicTransactionSubsystem))
    2.2 Satifaction Constraint
          mutual_require (ATM, CustomerValidationServer)
```

Table 5.22 Architecture Model in Disjunctive Normal Form (Abstract Component Level) in the UDSL for the Bank Example (4 disjunctives)

```
   Architecture Model in Disjunctive Normal Form (Abstract Component Level)

1. Disjunctive Normal Form
        Bank: one-of (BankCase1, BankCase2, BankCase3, BankCase4)

        BankCase1: all (ATM, CashierTerminal, CustomerValidationServer,
                        CashierValidationServer, TransactionServerManager,
                        EconomicTransactionServer)
        BankCase2: all (ATM, CashierTerminal, CustomerValidationServer,
                        CashierValidationServer, TransactionServerManager,
                        DeluxeTransactionServer, AccountDatabase)
        BankCase3: all (CashierTerminal, CashierValidationServer,
                        TransactionServerManager, EconomicTransactionServer)
        BankCase4: all (CashierTerminal, CashierValidationServer,
                        TransactionServerManager, DeluxeTransactionServer,
                        AccountDatabase)

2. Constraint Expression
        2.1 Default Constraint
                default (Bank : BankCase3)
```

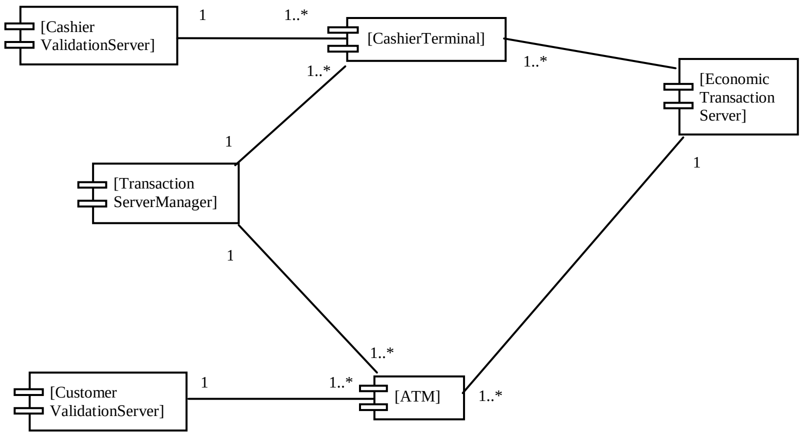


Figure 5.14 Component Diagram of BankCase1 for Banking Domain

5.3.3 Creating Sequence Diagrams

Sequence diagram is good for showing how use cases are carried out by appropriate components. Create one or more sequence diagrams to show how the autonomous components in the system interact with each other and with users. At least one sequence diagram should be created for each use case identified. It is possible there are variations in realizing a use case and there may be multiple ways to realize a use case as this is inherent in the development of a DCS family. In such a situation, a separate sequence diagram should be created for each alternative. During the creation of the sequence diagrams for each use case, also design the communication patterns of the function calls between components. A communication pattern shows the characteristic of parallelism of a function. The basic communication patterns include *one-way*, *two-way-synchronous* and *two-way-asynchronous*, which are discussed in Chapter 4. The information about communication patterns is not shown in the sequence diagram, but is summarized in the function summary of abstract components in Section 5.3.5. Figure 5.15 shows the sequence diagram of the *Deposit Money* use case for a cashier when *EconomicTransactionServer* is involved. A complete list of sequence diagrams is in Appendix C.

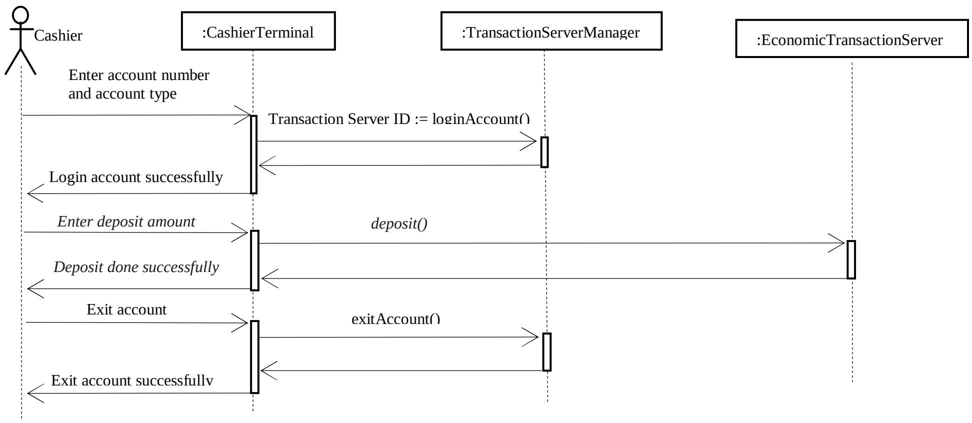


Figure 5.15 Sequence Diagram of Deposit Money (Case 1)

Table 5.23 CUCM at the Abstract Component Level in the UDSL for the Banking Domain Example

Critical Use Case Model at the Abstract Component Level
1. Commonality and Variation CriticalUseCaseModel: all (DepositMoney_Cashier, WithdrawMoney_Cashier, TransferMoney_Cashier) DepositMoney_Cashier: one-of (DepositMoneyCase1, DepositMoneyCase2) DepositMoneyCase1: path_c (CashierTerminal, DeluxeTransactionServer, AccountDatabase) DepositMoneyCase2: path_c (CashierTerminal, EconomicTransactionServer) WithdrawMoney_Cashier: one-of (WithdrawMoneyCase1, WithdrawMoneyCase2) WithdrawMoneyCase1: path_c (CashierTerminal, DeluxeTransactionServer, AccountDatabase) WithdrawMoneyCase2: path_c (CashierTerminal, EconomicTransactionServer) TransferMoney_Cashier: one-of (TransferMoneyCase1, TransferMoneyCase2) TransferMoneyCase1: path_c (CashierTerminal, DeluxeTransactionServer, AccountDatabase) TransferMoneyCase2: path_c (CashierTerminal, EconomicTransactionServer) 2. Constraint Expresssion 2.1 Default Constraint default (DepositMoney_Cashier : DepositMoneyCase2) default (WithdrawMoney_Cashier : WithdrawMoneyCase2) default (TransferMoney_Cashier : TransferMoneyCase2) 2.2 Satisfaction Constraint mutual_require (DepositMoneyCase1, WithdrawMoneyCase1, TransferMoneyCase1) mutual_require (DepositMoneyCase2, WithdrawMoneyCase2, TransferMoneyCase2)

5.3.4 Refining Critical Use Case Model to Abstract Component Level

From the sequence diagram and component diagram, summarize the communication path for each critical use case to refine the Critical Use Case Model CUCM) shown in Figure 5.5 and Table 5.8 to the abstract component level as shown in Table 5.23.

5.3.5 Identifying Component Interfaces and Communication Patterns

For each abstract component, two kinds of interfaces need to be identified, *required interfaces* and *provided interfaces*. The *provided interfaces* are those interfaces provided by a design feature to other design features. The *required interfaces* are those interfaces required by this design feature from other design features. In order to identify these interfaces, the first thing in this step is to summarize the actions, inputs and outputs of each component from the sequence diagrams for each abstract component. Table 5.24 shows an example of the summarization for the *TransactionServerManger*. A complete list of all summaries is in Appendix D.

The next step is to derive interfaces. This is the process of grouping related functions across abstract components. The procedure is based on the summary of actions, inputs and outputs for abstract components. Reference to the use case model and requirement model in the domain analysis stage is also a great help to derive meaningful interfaces. Each interface is documented in an interface description table, which consists of the precondition, postcondition, invariant, communication pattern, and description for each function in the interface. It also consists of variation of the interface. For this work, the variation is caused solely by the communication patterns. Communication patterns considered in this work include: one-way, two-way-synchronous and two-way-asynchronous, which are denoted as cp1, cp2s and cp2a, respectively. All these interfaces form the *Interface Model*. From the *Interface Model,* summarize the provided interfaces and required interfaces for each abstract component. The whole process is iterative and incremental, and usually needs prototyping.

Table 5.24 Function Summary for *TransactionManager* in the Banking Domain Example

TransactionServerManager			
Actions	Inputs	Outputs	Communication Pattern
loginAccount	Account Number, Account Type	Transaction Server ID	two-way-synchronous
exitAccount	Account Number, Account Type	NONE	two-way-synchronous
openAccount	Account Number, Account Type	Account Number, Account Type Transaction Server ID	two-way-synchronous
closeAccount	Account Number, Account Type	Transaction Server ID	two-way-synchronous

Table 5.25 Interface Description for *IAccountDatabase* in the Banking Domain Example

IAccountDatabase

1. Syntax

Account getAccount(String accountNumber, int accountType);
    Pre: values have been provided for the accountNumber and accountType.
    Post: if the specified account exits, return the account; otherwise return NULL.
    Invariant: accountNumber, accountType
    Communication Pattern: cp2s or cp2a
    Description: This function returns an account object as identified by the parameters. It returns null if the account specified does not exist.

void saveAccount(Account account);
    Pre: account is valid
    Post: the database has been updated appropriately.
    Invariant: account
    Communication Pattern: cp2s or cp2a
    Description: This function updates the account if it already exists; otherwise it adds an entry in the database for this new account.

void removeAccount(String accountNumber, int accountType);
    Pre: values have been provided for the account and accountType
    Post: the account specified is removed and the database has been updated appropriately
    Invariant: accountNumber, accountType
    Communication Pattern: cp2s or cp2a
    Description: This function removes the specified account if it exists; otherwise it does nothing.

2. Variation

IAccountDatabase: one-of (IAccountDatabaseCase1, IAccountDatabaseCase2)
IAccountDatabaseCase1: {cp2s}
IAccountDatabaseCase2: {cp2a}

3. Default Constraint

default (IAccountDatabase : IAccountDatabaseCase1)

For the banking domain example, following interfaces are identified to cover the functionality in the requirement model: *ICustomerManagement*, *IAccountManagement*, *ITransactionServerManager*, *IAccountDatabase*, and *IUserValidation*. Table 5.26 is an example of an interface description. The expression, *IAccountDatabaseCase1: {cp2s}*, means that all functions in *IAccountDatabaseCase1* are two-way-synchronous. A complete list of all interface descriptions for the banking domain example is in Appendix E. The provided interfaces and required interfaces for each abstract component are summarized by consulting the sequence diagrams and are shown in Table 5.26, which is actually a summary of the abstract components at the function/interface level. Table 5.26 can be expressed in the UDSL as shown in Table 5.27. Then it is expressed in disjunctive normal form in Table 5.28 and Table 5.29 with consideration of all the variations of interfaces, which forms the Abstract Component Interface Model (ACIM) in the UGDM.

Table 5.26 Provided Interfaces and Required Interfaces of Abstract Components for the Banking Domain Example

Abstract Components	Provided Interface	Required Interface
CashierTerminal	ICustomerManagement IAccountManagement IValidation	ICustomerManagement IAccountManagement ITransactionServerManager IValidation
ATM	IAccountManagement IValidation	IAccountManagement ITransactionServerManager IValidation
TransactionServerManager	ITransactionServerManager	NONE
EconomicTransactionServer	ICustomerManagement IAccountManagement	NONE
DeluxeTransactionServer	ICustomerManagement IAccountManagement	IAccountDatabase
AccountDatabase	IAccountDatabase	NONE
CashierValidationServer	IValidation	NONE
CustomerValidationServer	IValidation	NONE

Table 5.27 Abstract Components at Functional/Interface Level in UDSL for the Banking Domain Example

Abstract Components at Functional/Interface Level
• Design Feature Expression interface (CashierTerminal: provided_interface (ICustomerManagement, IAccountManagement, IValidation), required_interface (ICustomerManagement, IAccountManagement, ITransactionServerManager, IValidation)) interface (ATM: provided_interface (IAccountManagement, IValidation), required_interface (IAccountManagement, ITransactionServerManager, IValidation)) interface (CashierValidationServer: provided_interface (IValidation), required_interface (NONE)) interface (CustomerValidationServer: provided_interface (IValidation), required_interface (NONE)) interface (TransactionServerManager: provided_interface (ITransactionServerManager), required_interface (NONE)) interface (EconomicTransactionServer: provided_interface (IAccountManagement, ICustomerManagement), required_inteface (NONE)) interface (DeluxeTransactionServer: provided_interface (IAccountManagement, ICustomerManagement), required_interface (IAccountDatabase)) interface (AccountDatabase: provided_interface (IAccountDatabase), required_interface (NONE))

Table 5.28 ACIM in the UDSL for the Banking Domain Example

Abstract Component Interface Model
1. Disjunctive Normal Form CashierTerminal: CashierTerminalCase1 ATM: ATMCase1 CashierValidationServer: CashierValidationServerCase1 CustomerValidationServer: CustomerValidationServerCase1 TransactionServerManager: TransactionServerManagerCase1 EconomicTransactionServer: EconomicTransactionServerCase1 DeluxeTransactionServer: one-of (DeluxeTransaxtionServerCase1, DeluxeTransactionServerCase2) AccountDatabase: one-of (AccountDatabaseCase1, AccountDatabaseCase2) interface (CashierTerminalCase1: provided_interface (ICustomerManagementCase1, IAccountManagementCase1), required_interface (ICustomerManagementCase1, IAccountManagementCase1, ITransactionServerManagerCase1, IValidationCase1)) interface (ATMCase1: provided_interface (IAccountManagementCase1), required_interface (IAccountManagementCase1, ITransactionServerManagerCase1, IValidationCase1)) interface (CashierValidationServerCase1: provided_interface (IValidationCase1), required_interface (NONE)) interface (CustomerValidationServerCase1: provided_interface (IValidationCase1), required_interface (NONE)) interface (TransactionServerManagerCase1: provided_interface (ITransactionServerManagerCase1), required_interface (NONE)) interface (EconomicTransactionServerCase1: provided_interface (IAccountManagementCase1, ICustomerManagementCase1), required_inteface (NONE)) interface (DeluxeTransaxtionServerCase1: provided_interface (IAccountManagementCase1, ICustomerManagementCase1), required_interface (IAccountDatabaseCase1)) (Continued in Table 5.29)

Table 5.29 ACIM in the UDSL for the Banking Domain Example
(Continued from Table 5.28)

Abstract Component Interface Model
(Continued from Table 5.28) interface (DeluxeTransactionServerCase2: provided_interface (IAccountManagementCase1, ICustomerManagementCase1), required_interface (IAccountDatabaseCase2)) interface (AccountDatabaseCase1: provided_interface (IAccountDatabaseCase1), required_interface (NONE)) interface (AccountDatabaseCase2: provided_interface (IAccountDatabaseCase2), required_interface (NONE)) 2. Constraint Expression 2.1 Default Constraint default (DeluxeTransactionServer : DeluxeTransactionServerCase1) default (AccountDatabase : AccountDatabaseCase1) 2.2 Satisfaction Constraint mutual_require (DeluxeTransactionServerCase1, AccountDatabaseCase1) mutual_require (DeluxeTransactionServerCase2, AccountDatabaseCase2)

Next, from the ACIM in Table 5.28 and Table 5.29, derive a mapping for an abstract component from the component level to the function/interface level to impose the default constraints. The mapping is shown in Table 5.30.

Table 5.30 Mapping of Abstract Component from Component Level to Function/Interface Level in the UDSL for the Banking Domain Example

Mapping of Abstract Component from Component Level to Functional/Interface Level
map (CashierTerminal: CashierTerminalCase1) map (ATM: ATMCase1) map (CashierValidationServer: CashierValidationServerCase1) map (CustomerValidationServer: CustomerValidationServerCase1) map (TransactionServerManager: TransactionServerManagerCase1) map (EconomicTransactionServer: EconomicTransactionServerCase1) map (DeluxeTransactionServer: DeluxeTransaxtionServerCase1) map (AccountDatabase: AccountDatabaseCase1)

5.3.6 Refining Critical Use Case Model to Function/Interface Level

This step refines the critical use case model at the component level created in Section 5.3.4 to the function/interface level by consulting the ACIM and the result is shown in Table 5.31 and Table 5.32. At this level, each critical use case is expressed as a path of function calls. The communication for each function call is also stated. Section 4.2.2.4 has more information about this expression. The model at this level is crucial for deriving the QoS Composition and Decomposition Model as described in Section 5.3.10. Table 5.33 shows the normalized expression for the critical use case model at the function/interface level. The disjunctive normal form of this critical use case model in Table 5.34 is derived from the normalized expression.

Table 5.31 CUCM at Function/Interface Level for the Banking Domain Example

Critical Use Case Model (Function/Interface Level)
1. Use Case Expression CriticalUseCase: all (DepositMoney_Cashier, WithdrawMoney_Cashier, TransferMoney_Cashier) DepositMoney_Cashier: one-of (DepositMoneyCase1, DepositMoneyCase2) DepositMoneyCase1: one-of (DepositMoneyCase1_1, DepositMoneyCase1_2) DepositMoneyCase1_1: path_f(CashierTerminal.deposit[cp2s], DeluxeTransactionServer.deposit[cp2s], AccountDatabase.getAccount[cp2s], AccountDatabase.saveAccount[cp2s]) DepositMoneyCase1_2: path_f (CashierTerminal.deposit[cp2s], DeluxeTransactionServer.deposit[cp2s], AccountDatabase.getAccount[cp2a], AccountDatabase.saveAccount[cp2a]) DepositMoneyCase2: path_f (CashierTerminal.deposit[cp2s], EconomicTransactionServer.deposit[cp2s]) WithdrawMoney_Cashier: one-of (WithdrawMoneyCase1, WithdrawMoneyCase2) WithdrawMoneyCase1: one-of (WithdrawMoneyCase1_1, WithdrawMoneyCase1_2) WithdrawMoneyCase1_1: path_f (CashierTerminal.withdraw[cp2s], DeluxeTransactionServer.withdraw[cp2s], AccountDatabase.getAccount[cp2s], AccountDatabase.saveAccount[cp2s]) WithdrawMoneyCase1_2: path_f (CashierTerminal.withdraw[cp2s], DeluxeTransactionServer.withdraw[cp2s], AccountDatabase.getAccount[cp2a], AccountDatabase.saveAccount[cp2a]) WithdrawMoneyCase2: path_f (CashierTerminal.transfer[cp2s], EconomicTransactionServer.transfer[cp2s]) TransferMoney_Cashier: one-of (TransferMoneyCase1, TransferMoneyCase2) TransferMoneyCase1: one-of (TransferMoneyCase1_1, TransferMoneyCase1_2) TransferMoneyCase1_1: path_f (CashierTerminal.transfer[cp2s], DeluxeTransactionServer.transfer[cp2s], AccountDatabase.getAccount[cp2s], AccountDatabase.saveAccount[cp2s]) TransferMoneyCase1_2: path_f (CashierTerminal.transfer[cp2s], DeluxeTransactionServer.transfer[cp2s], AccountDatabase.getAccount[cp2a], AccountDatabase.saveAccount[cp2a]) TransferMoneyCase2: path_f (CashierTerminal.transfer[cp2s], EconomicTransactionServer.transfer[cp2s]) (Continued in Table 5.32)

Table 5.32 CUCM at Function/Interface Level for the Banking Domain Example (Continued from Table 5.31)

Critical Use Case Model (Function/Interface Level)
(Continued from Table 5.31) 2. Constraint Expression 2.1 Default Constraint default (DepositMoney_Cashier : DepositMoneyCase2) default (WithdrawMoney_Cashier : WithdrawMoneyCase2) default (TransferMoney_Cashier : TransferMoneyCase2) 2.2 Satisfaction Constraint mutual_require (DepositMoneyCase1_1, WithdrawMoneyCase1_1, TransferMoneyCase1_1) mutual_require (DepositMoneyCase1_2, WithdrawMoneyCase1_2, TransferMoneyCase1_2) mutual_require (DepositMoneyCase2, WithdrawMoneyCase2, TransferMoneyCase2)

Table 5.33 Normalized Expression of CUCM at Function/Interface Level for the Banking Domain Example

Normalized Expression of Critical Use Case Model (Function/Interface Level)
1. Use Case Expression CriticalUseCase: all (one-of (one-of(DepositMoneyCase1_1, DepositMoneyCase1_2), DepositMoneyCase2), one-of (one-of(WithdrawMoneyCase1_1, WithdrawMoneyCase1_2), WithdrawMoneyCase2), one-of (one-of(TransferMoneyCase1_1, TransferMoneyCase1_2), TransferMoneyCase2)) DepositMoneyCase1_1: path_f(CashierTerminal.deposit[cp2s], DeluxeTransactionServer.deposit[cp2s], AccountDatabase.getAccount[cp2s], AccountDatabase.saveAccount[cp2s]) DepositMoneyCase1_2: path_f (CashierTerminal.deposit[cp2s], DeluxeTransactionServer.deposit[cp2s], AccountDatabase.getAccount[cp2a], AccountDatabase.saveAccount[cp2a]) DepositMoneyCase2: path_f (CashierTerminal.deposit[cp2s], EconomicTransactionServer.deposit[cp2s]) WithdrawMoneyCase1_1: path_f (CashierTerminal.withdraw[cp2s], DeluxeTransactionServer.withdraw[cp2s], AccountDatabase.getAccount[cp2s], AccountDatabase.saveAccount[cp2s]) WithdrawMoneyCase1_2: path_f (CashierTerminal.withdraw[cp2s], DeluxeTransactionServer.withdraw[cp2s], AccountDatabase.getAccount[cp2a], AccountDatabase.saveAccount[cp2a]) WithdrawMoneyCase2: path_f (CashierTerminal.transfer[cp2s], EconomicTransactionServer.transfer[cp2s]) TransferMoneyCase1_1: path_f (CashierTerminal.transfer[cp2s], DeluxeTransactionServer.transfer[cp2s], AccountDatabase.getAccount[cp2s], AccountDatabase.saveAccount[cp2s]) TransferMoneyCase1_2: path_f (CashierTerminal.transfer[cp2s], DeluxeTransactionServer.transfer[cp2s], AccountDatabase.getAccount[cp2a], AccountDatabase.saveAccount[cp2a]) TransferMoneyCase2: path_f (CashierTerminal.transfer[cp2s], EconomicTransactionServer.transfer[cp2s]) 2. Constraint Expression 2.1 Default Constraint default (CriticalUseCase : all (DepositMoneyCase2, WithdrawMoneyCase2, TransferMoneyCase2)) 2.2 Satisfaction Constraint mutual_require (DepositMoneyCase1_1, WithdrawMoneyCase1_1, TransferMoneyCase1_1) mutual_require (DepositMoneyCase1_2, WithdrawMoneyCase1_2, TransferMoneyCase1_2) mutual_require (DepositMoneyCase2, WithdrawMoneyCase2, TransferMoneyCase2)

Table 5.34 CUCM in Disjunctive Normal Form at Function/Interface Level in the UDSL for the Banking Domain Example

Disjunctive Normal Form of the Critical Use Case Model (Function/Interface Level)

1. Disjunctive Normal Form
    CriticalUseCase: one-of (CriticalUseCase1, CriticalUseCase2, CriticalUseCase3)
    CriticalUseCase1: all (DepositMoneyCase1_1, WithdrawMoneyCase1_1, TransferMoneyCase1_1)
    CriticalUseCase2: all (DepositMoneyCase1_2, WithdrawMoneyCase1_2, TransferMoneyCase1_2)
    CriticalUseCase3: all (DepositMoneyCase2, WithdrawMoneyCase2, TransferMoneyCase2)
 	
    DepositMoneyCase1_1: path_f(CashierTerminal.deposit[cp2s], DeluxeTransactionServer.deposit[cp2s], AccountDatabase.getAccount[cp2s], AccountDatabase.saveAccount[cp2s])
    DepositMoneyCase1_2: path_f (CashierTerminal.deposit[cp2s], DeluxeTransactionServer.deposit[cp2s], AccountDatabase.getAccount[cp2a], AccountDatabase.saveAccount[cp2a])
    DepositMoneyCase2: path_f (CashierTerminal.deposit[cp2s], EconomicTransactionServer.deposit[cp2s])
    WithdrawMoneyCase1_1: path_f (CashierTerminal.withdraw[cp2s], DeluxeTransactionServer.withdraw[cp2s], AccountDatabase.getAccount[cp2s], AccountDatabase.saveAccount[cp2s])
    WithdrawMoneyCase1_2: path_f (CashierTerminal.withdraw[cp2s], DeluxeTransactionServer.withdraw[cp2s], AccountDatabase.getAccount[cp2a], AccountDatabase.saveAccount[cp2a])
    WithdrawMoneyCase2: path_f (CashierTerminal.transfer[cp2s], EconomicTransactionServer.transfer[cp2s])
    TransferMoneyCase1_1: path_f (CashierTerminal.transfer[cp2s], DeluxeTransactionServer.transfer[cp2s], AccountDatabase.getAccount[cp2s], AccountDatabase.saveAccount[cp2s])
    TransferMoneyCase1_2: path_f (CashierTerminal.transfer[cp2s], DeluxeTransactionServer.transfer[cp2s], AccountDatabase.getAccount[cp2a], AccountDatabase.saveAccount[cp2a])
    TransferMoneyCase2: path_f (CashierTerminal.transfer[cp2s], EconomicTransactionServer.transfer[cp2s])
 	
2. Constraint Expression
    2.1 Default Contraint
        default (CriticalUseCase : CriticalUseCase3)

5.3.7 Refining Architecture Model in Disjunctive Normal Form from Component Level to Function/Interface Level

This step refines the Architecture Model in Disjunctive Normal Form (AMDNF) at the component level (shown in Table 5.22) developed in Section 5.3.2 into the function/interface level by consulting the ACIM. The result is shown in Table 5.35. The normalization process takes into account the satisfaction constraints in the ACIM. There are totally 6 disjunctives in the AMDNF at function/interface level for the banking domain example.

Table 5.35 AMDNF at Function/Interface Level
in the UDSL for the Banking Domain Example

Disjunctive Normal Form of Architecture Model (Function/Interface Level)
1. Disjunctive Normal Form Bank: one-of (BankCase1, BankCase2, BankCase3, BankCase4) BankCase1: BankCase1_1 BankCase2: one-of (BankCase2_1, BankCase2_2) BankCase3: BankCase3_1 BankCase4: one-of (BankCase4_1, BankCase4_2) BankCase1_1: all (ATMCase1, CashierTerminalCase1, CustomerValidationServerCase1, CashierValidationServerCase1, TransactionServerManagerCase1, EconomicTransactionServerCase1) BankCase2_1: all (ATMCase1, CashierTerminalCase1, CustomerValidationServerCase1, CashierValidationServerCase1, TransactionServerManagerCase1, DeluxeTransactionServerCase1, AccountDatabaseCase1) BankCase2_2: all (ATM, CashierTerminalCase1, CustomerValidationServerCase1, CashierValidationServerCase1, TransactionServerManagerCase1, DeluxeTransactionServerCase2, AccountDatabaseCase2) BankCase3_1: all (CashierTerminalCase1, CashierValidationServerCase1, TransactionServerManagerCase1, EconomicTransactionServerCase1) BankCase4_1: all (CashierTerminal, CashierValidationServer, TransactionServerManager, DeluxeTransactionServerCase1, AccountDatabaseCase1) BankCase4_2: all (CashierTerminalCase1, CashierValidationServerCase1, TransactionServerManagerCase1, DeluxeTransactionServerCase2, AccountDatabaseCase2) 2. Default Constraint default (BankCase2 : BankCase2_1) default (BankCase4 : BankCase4_1)

From Table 5.22 (AMDNF at the component level) and Table 5.35 (AMDNF at the function/interface level), derive a mapping for the AMDNF from the component level to the function/interface level to impose the default constraints as shown in Table 5.35. The mapping is shown in Table 5.36.

Table 5.36 Mapping of AMDNF from Component Level to Function/Interface Level in the UDSL for the Banking Domain Example

Mapping of AMDNF from Component Level to Function/Interface Level
map (BankCase1: BankCase1_1) map (BankCase2: BankCase2_1) map (BankCase3: BankCase3_1) map (BankCase4: BankCase4_1)

5.3.8 Mapping Architecture Model in Disjunctive Normal Form to Critical Use Case Model (Function/Interface Level)

This step is to create a mapping from the Architecture Model in Disjunctive Normal Form (AMDNF) at the function/interface level to the Critical Use Case Model (CUCM) in disjunctive normal form at the function/interface level, i.e., a mapping from Table 5.35 to Table 5.34 for the banking domain example. The mapping is based on the component diagrams developed in Section 5.3.2 and the sequence diagrams developed in Section 5.3.3. The components participate in the realization of the critical use cases which form a case of the CUCM must be among the components in a case of the AMDNF. The mapping from the AMDNF to the CUCM is unique. However, more than one case of the AMDNF can be mapped to one case of the CUCM. This mapping is a connection relating the system architecture to the system QoS. The mapping for the banking domain example is shown in Table 5.37.

Table 5.37 AMDNF and CUCM Mapping (Function/Interface Level) for the Banking Domain Example

AMDNF and CUCM Mapping (Function/Interface Level)
mapping (BankCase1_1 : CriticalUseCase3) mapping (BankCase2_1 : CriticalUseCase1) mapping (BankCase2_2 : CriticalUseCase2) mapping (BankCase3_1 : CriticalUseCase3) mapping (BankCase4_1 : CriticalUseCase1) mapping (BankCase4_2 : CriticalUseCase2)

5.3.9 Creating Abstract Component Model

The Abstract Component Model (ACM) consists of the UMM specifications for all the abstract components in a DCS domain. The UMM specification is described in detail in Section 3.3. A full list of UMM specifications for all the abstract components in the banking domain example is in Appendix F.

5.3.10 Creating QoS Composition and Decomposition Model

The QoS Composition and Decomposition Model (QCDM) for a domain consists of all the composition and decomposition rules for the identified QoS parameters on each critical use case. Table 5.38 shows the QoS composition and decomposition meta-rules used in the banking domain example. These rules are domain independent, and are called meta-rules to distinguish them from the rules derived from them for critical use cases of a specific DCS domain. Details about the QoS composition and decomposition meta-rules are in [SUN02, SUN03].

The QCDM for a specific DCS domain is a direct application of the QoS composition and decomposition mea-rules. The application of the meta-rules in Table 5.38 on *throughput* and *endToEndDelay* for all the critical use cases of the banking domain example results in the QoS composition and decomposition rules for the banking domain which are organized into four sets and are listed in Appendix G.

From the QoS composition and decomposition rules for the banking domain example, the QoS Composition and Decomposition Model (QCDM) for each case of the Critical Use Case Model (CUCM) in disjunctive normal form can be derived. The results are shown in Appendix H. The QoS composition and decomposition model for each bank instance is then determined when the architecture model is determined. The connection between these two is done through the mapping developed in Section 5.3.8.

Table 5.38 QoS Composition and Decomposition Meta-Rules Used
in the Banking Domain Example

QoS Composition and Decomposition Meta-Rules

Notations:
[CriticalUseCaseModelCase]: a case of a critical use case model at disjunctive normal form
{CriticalUseCases}: all critical use cases in a case of a critical use case model
[CriticalUseCase]: one critical use cases in a case of a critical use case model
<CriticalUseCase>: all function calls in a critical use case

1. QoS Composition Rules:
   - 1.1 Composition rules for *throughput*
     - 1.1.1 System_througput = [CriticalUseCaseModelCase]_throughput
     - 1.1.2 [CriticalUseCaseModelCase]_throughput = min ({CriticalUseCases}_throughput)
     - 1.1.3 Let [CriticalUseCase]: path ($CALL_1$, $CALL_2$, …, $CALL_N$)

       $T_1 = CALL_N\_throughput$

       $$\begin{cases} T_n = \min(CALL_{N-n+1}\_throughput, T_{n-1}), \text{ if } CALL_{N-n+2} \text{ is asynchronous} \\ 1/T_n = 1/CALL_{N-n+1}\_throughput + 1/T_{n-1}), \text{ if } CALL_{N-n+2} \text{ is synchronous} \end{cases}$$

       [CriticalUseCase]_throughput = T_N
   - 1.2 Composition rules for *endToEndDelay*
     - 1.2.1 System_endToEndDelay = [CriticalUseCaseModelCase]_endToEndDelay
     - 1.2.2 [CriticalUseCaseModelCase]_endToEndDelay = max ({CriticalUseCases}_endToEndDelay)
     - 1.2.3 [CriticalUseCase]_endToEndDelay = sum (<CriticalUseCase>_endToEndDelay)

2. QoS Decomposition Rules:
   - 2.1 Decomposition rules for *throughput*

     [CriticalUseCaseModelCase]_throughput > System_throughput
     {CriticalUseCases}_throughput > System_throughput
     <CriticalUseCase>_throughput > System_throughput
   - 2.2 Decomposition rules for *endToEndDelay*

     [CriticalUseCaseModelCase]_endToEndDelay < System_endToEndDelay
     {CriticalUseCases}_endToEndDelay < System_endToEndDelay
     <CriticalUseCase>_endToEndDelay < System_endToEndDelay

5.4 Ordering Language Design

An ordering language is anther important artifact in the UGDP. The ordering language is the interface that the application engineers (users) employ to order concrete systems from a DCS family. This language is a kind of domain specific language. It can be textual, tabular, graphical, or even natural-language-like.

The UDSL itself can be viewed as an ordering language. In this sense, the UDSL defined a layered DSL which can specify a system to different levels of detail. Three levels can be identified in the UDSL as an ordering language in the UA process: level of system architecture, level of functionality (including communication patterns) and level of the QoS. These three levels are inherent in the UDSL. During the UGDP, transformations and mappings are developed for various models, thus the UDSL is hierarchical and is powerful enough to express to the level of detail necessary for the application programmers. In order to use the UDSL as an ordering language, an application engineer must study the UGDM for a DCS family and becomes a domain expert in some degree.

The tabular ordering language is an attractive method to order a system. It is simple to use when compared with the UDSL. In this language, the possible systems in a system family are categorized and available options are provided. Here is an analog from the real world. When ordering a car from a dealer, there is no need to describe to the dealer to the great detail about what kind of car is needed. There is no need to describe to the extreme detail like suspension, trunklet, etc. Cars are ordered by stating the model, the trim and the options. The same can be done in generative programming. In the banking domain example, we can state the class of the bank, options and desired QoS to order a bank as shown in Table 5.39. Or simply say "get me a bank", in which case, the Basic Bank is returned as default.

Table 5.39 Tabular Ordering Language for
the Banking Domain Example

	BasicBank	AdvancedBank	SuperBank
User Terminal			
ATM		o	•
CashierTerminal	•	•	•
System QoS			
endToEndDelay	o (2000)	o (1500)	o (1000)
throughput	o (500)	o (900)	o (1500)
Legend: • standard requirements o optional requirements () default values			

Table 5.40 Mapping Rules for the Tabular Ordering Language of the Banking Domain Example

Mapping Rules for the Tabular Ordering Language of the Banking Domain Example
If no ATM and system throughput <= 650 operations/second, map to BankCase3
Else if no ATM and system throughput > 650 operations/second, map to BankCase4
Else if 1 ATM and system throughput <= 800 operations/second , map to BankCase1
Else if 1 ATM and system throughput > 800 operations/second , map to BankCase2
Else if the copy number of ATM is greater than or equal to 2, map to BankCase2

Next a mapping from the tabular ordering language to the UGDM described in the UDSL needs to be designed. There are no rules how the mapping should be done. The mapping is domain dependent and it changes overtime just like the “car ordering language” which changes every year when the new car models are available. For the banking domain example, the simple mapping rules are designed to translate the tabular ordering language into the Architecture Model in Disjunctive Normal Form at the component level. The mapping rules are shown in Table 5.40.

Natural-language-like ordering language is also very attractive and it is very flexible. However, it is more difficult to implement and requires natural language processing support. Domain specific order information is needed by a natural language processor in order to process any query in that domain. The work on natural language processing to support the UniFrame is carried out by University of Alabama at Birmingham [LEE02, LEE02a], a collaborator of UniFrame research. An example of an order in natural-language-like format in the banking domain example is: “Generate a bank system with 1 ATM and 2 cashier terminals. The turn around time is less than 2000 microseconds, and the throughput is greater than 500 operations/second”.

This chapter presents in detail the UGDP for developing the UGDM for a selected DCS domain. The UGDP is an iterative and incremental process. The UGDM evolves during iterations of the UGDP. This is the best way to achieve a stable and mature UGDM for a specific DCS domain. In next chapter, the UniFrame System Generation Infrastructure (USGI) that uses the UGDM and implements the UGDM processing logic is provided.

6. The UNIFRAME SYSTEM GENERATION INFRASTRUCTURE (USGI)

Chapter 4 describes the UGDM which captures the common and variable properties of a DCS family. The UGDM takes into consideration of the importance of QoS in order to generate a QoS-aware DCS. Chapter 5 describes the UGDP, which is a process for creating a UGDM. Presented in this chapter is the UniFrame System Generation Infrastructure (USGI), which is the third part of the USGPF. The USGI is an infrastructure for realizing system-level generative programming. The description of the USGI in this chapter focuses on the high-level design, workflow modeling, the algorithm and the interaction of modules that comprise the USGI. These descriptions are at a conceptual level and are not tied to any software or technology that may implement the architecture.

6.1 Overview of the USGI Architecture

The USGI helps to automatically generate a DCS from a DCS family by integrating heterogeneous distributed software components based on a UGDM. It is not intended for component code generation. It reflects the application engineering phase in the component based software engineering process and directly uses the UGDM created during domain engineering.

The architecture of the USGI is shown in Figure 6.1. It consists of several modules. Here is the brief description of the functionality of each module in this framework.

- *URDS*: This module is responsible for the active component management. It dynamically discovers and manages the heterogeneous software components deployed over the network by component developers. It also assists in the finding of the concrete components for the abstract components required by the System Generator which is discussed below.

- *Wrapper and Glue Generator*: This module is responsible for creating the necessary wrapper and glue code to bridge heterogeneous distributed software components. The glue code also contains necessary instrumentations to compute the system QoS for the integrated system, which is the part of the dynamic system QoS validation.
- *UGDM Knowledgebase (UGDMKB)*: The module stores the UGDM and provides information about the UGDM to other modules in the framework. The module can be implemented as relational database tables or libraries, or both. For example, the QCDM can be implemented as a library and other models can be implemented as tables in a relational database.
- *UGDMKB Builder Terminal*: This is the module that provides the user interface to the software engineers who are responsible for the development and maintenance of the UGDM and the UGDMKB.
- *UGDMKB Generator*: This module is responsible for creating the UGDM and represents the UGDM in databases and/or libraries. This module automates the UGDP process to the extent feasible.
- *Application Programmer Terminal*: This is the module that provides the user interface to the application programmers or system assemblers and enables them to generate a DCS.
- *Order Processor*: This module is responsible for determining a DCS architecture instance from a DCS family that satisfies the system requirements provided by the application programmers or system assemblers according to the UGDM. A natural language processor may assist to process natural language-like orders using Two-Level Grammar (TLG) [LEE02, LEE02a].
- *System Generator*: This module is responsible for generating a DCS from a DCS family based on the UGDM. The *System Generator* implements the processing logic of the UGDM. In the USGI design, the UGDM is separated from the

processing logic of the UGDM. The merit of this approach is that as the UGDM evolves, the only thing that needs to be updated and maintained is the UGDMKB.

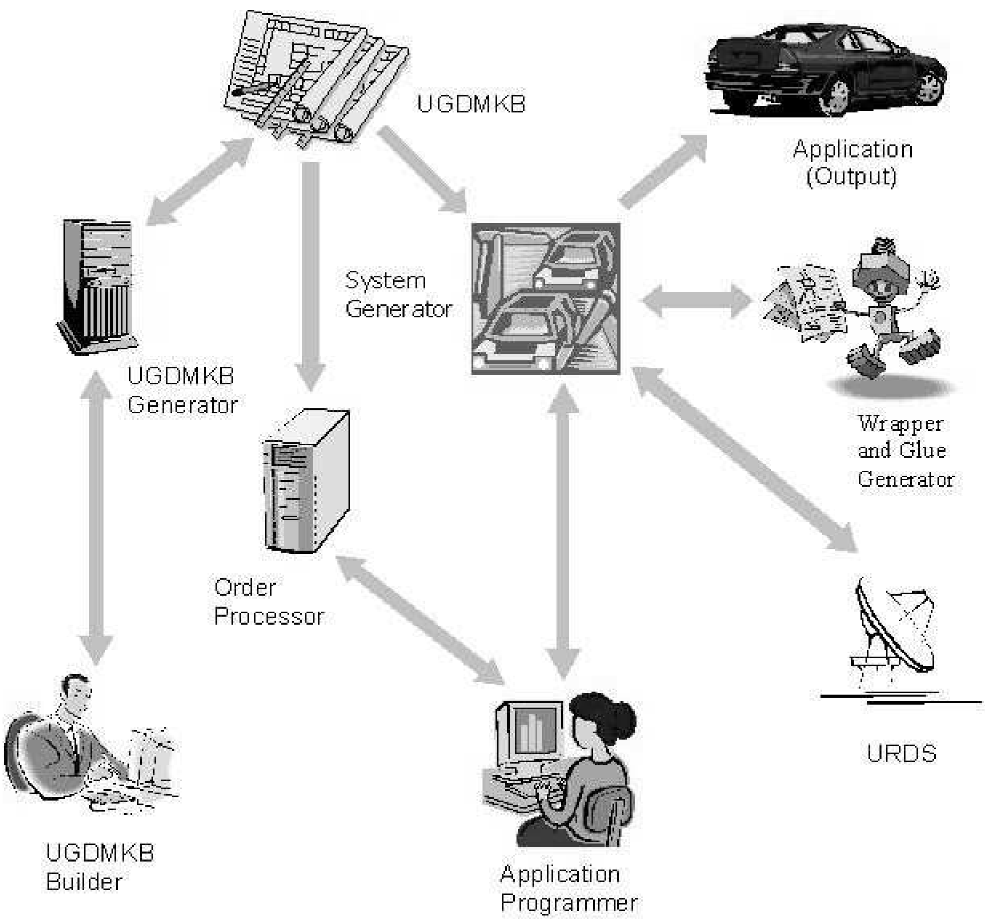


Figure 6.1 USGI Architecture

The detailed algorithms for each module are discussed in Section 6.3. The next section presents the dynamic modeling of the USGI workflow which shows the overall functionality of the framework, the role of each module in it and how the modules interact with each other to achieve the functionality of the USGI.

6.2 Modeling the USGI Workflow

The UML [BOO98, OMG03] modeling techniques proposed by Grady Booch and his colleagues are used to model the dynamic view of the USGI. The overall functionality of the USGI is modeled in an activity diagram shown in Figure 6.2. The interactions between each module in the framework are demonstrated by the object flow in Figure 6.3.

6.2.1 USGI Activity Diagram

An activity diagram [BOO98, OMG03] shows the flow from one activity to anther within a system. The diagram shows a set of activities, the sequential or branching flow from activity to activity in a system. The diagram illustrates the dynamic view of a system. Activity diagrams are especially important in modeling the functionality of a system. They model the system as a whole.

The major functionality of the USGI is to support the application engineering with generative programming to create a QoS-aware DCS from the available heterogeneous distributed software components which are geographically dispersed over the network. The major activities associated with this purpose include: gather system requirements, determine the required *component types*, called *abstract components*; these two terms are used interchangeably in this work), search the existing concrete components for the required abstract components, select a set of concrete components to assemble a DCS, determine the adapters which are required to bridge heterogeneous software components, validate the system QoS (both statically by the QoS composition and decomposition rules, and dynamically by the system behavior sampling which applies event grammars), integrate and deploy a system, and generate its UniFrame description which is an ongoing effort. The flow between these major activities is shown in Figure 6.2.

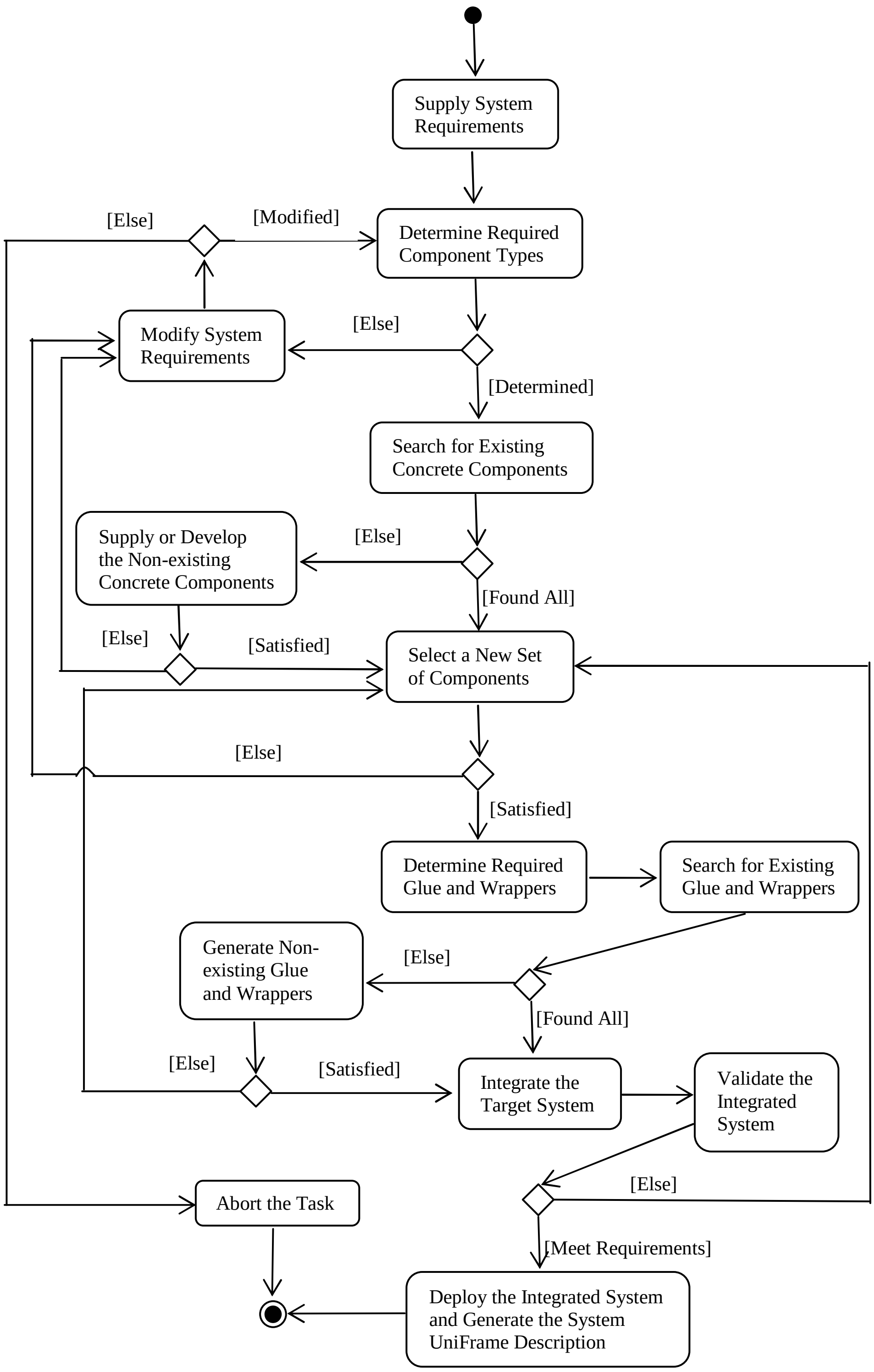


Figure 6.2 USGI Activity Diagram

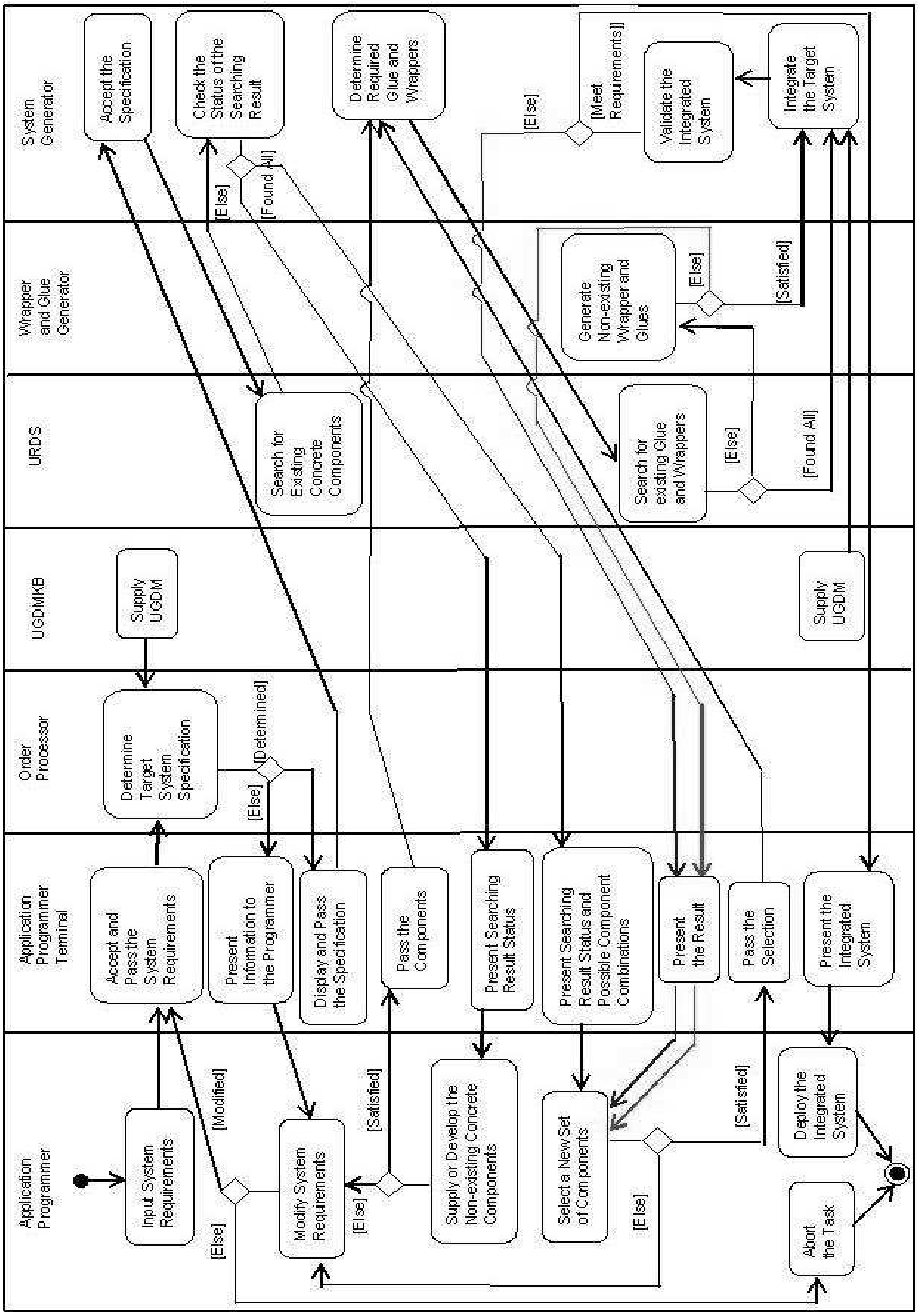


Figure 6.3 USGI Object Flow

6.2.2 USGI Object Flow

The object flow [BOO98, OMG03] is a special activity diagram that includes participating objects (modules in the USGI). It emphasizes the flow of control among different modules and shows the dependency relationships between them. Two kinds of relationships can be shown: the kinds of objects that have primary responsibility for performing an action and other objects whose values are used or determined by the action. The object flow partitions activities in an activity diagram into groups, each group representing a business process (module) that is responsible for those activities. In the UML, each group is called a swimlane. Swimlanes are a kind of package for organizing responsibility for activities. Thus, a swimlane specifies a locus of activities. Every activity belongs to exactly one swimlane, but transitions may cross lanes.

In the USGI, there is one swimlane for each module in the object flow, which is shown in Figure 6.3. The USGI object flow is derived from the USGI activity diagram by partitioning the activity diagram into swimlanes with the participating modules. It reflects all the activities in the USGI activity diagram while zooming into most of the activities. Thus, it reflects more detail about the flow and control information than the USGI activity diagram.

6.3 Modules of USGI

This section describes the functionality of each module in the USGI architecture, with emphasis on the *System Generator* module. Some modules in the USGI are results from other members of the UniFrame research, such as the URDS [SIR02], and the *Wrapper and Glue Generator* [CAO02, ZHA02]. This section provides only a brief description about this kind of modules.

6.3.1 Data Structures Used in Algorithms in Modules of USGI

Tables 6.1, 6.2 and 6.3 show the data structures used in the algorithms by various modules of the USGI, which are described in the following sections.

Table 6.1 Data Structure for Algorithms in System Generator

AbstractComponent *abstractComponent*	A data structure that holds the UMM description of an abstract component.
AdapterType *adapterType*	A data structure that defines an adapter type. It consists of a bridge type for two different component models and two component types that need to be bridged and their corresponding component models.
ConcreteComponent *concreteComponent,* *adapterComponent*	A data structure that holds the UMM description of a concrete component. An adapter itself is a component.
Hash Table *availableConcreteComponentTable*	A mapping between component types and the corresponding list of *available* concrete components. The names of component types serve as the keys for this mapping.
Hash Table *selectedConcreteComponentTable*	A mapping between component types and the corresponding list of *selected* concrete components. The names of component types serve as the keys for this mapping.
Hash Table *availableAdapterTable*	A mapping between adapter types and the corresponding list of available instances. The names of adapters serve as the keys for this mapping.
Hash Table *resultTable*	A mapping between component IDs and their corresponding detailed UniFrame Specifications. A data structure used in the URDS.
List *systemBluePrintList*	A list of instances of *systemBluePrint* for corresponding target system instances.

Table 6.2 Data Structure for Algorithms in System Generator
(Continued from Table 6.1)

List *requiredAbstractComponentList*	A list of all the required abstract components for a system specification.
List *availableConcreteComponentList*	A list of available concrete components for an abstract component.
List *selectedConcreteComponentList*	A list of selected concrete components for an abstract component.
List *adapterTypeList*	A list of required adapter types for a set of selected concrete components.
List *availableAdapterList*	A list of available adapter instances for an adapter type.
List *postprocessingCollaboratorList*	A list of post-processing collaborators of an abstract component.
QoSCompositionModel *qosCompositionModel*	A data structure or a library that contains the QoS Composition Model for a domain.
QueryBean *queryBean*	A data structure that holds a query about an abstract component. It is passed from the System Generator to the URDS.
QueryManager *queryManager*	A controller component in the URDS framework that interfaces other modules in the USGI.
Queue *selectedConcreteComponentTableQueue*	A queue that contains instances of possible *selectedConcreteComponentTable*. Each table is an instance of the potential target system.

Table 6.3 Data Structure for Algorithms in System Generator
(Continued from Table 6.2)

SystemBluePrint *systemBluePrint*	A data structure that contains detailed information about a system blueprint for a target system instance. It includes the related system specification, the selected concrete components, the necessary adapters, QoS validation results, etc.
SystemQoS *expectedSystemQoS* *staticSystemQoS* *dynamicSystemQoS*	A data structure that holds values of the system QoS. It can hold the expected system QoS, the static/predicted system QoS or the dynamic system QoS under different circumstances.
SystemSpecification *systemSpecification*	A data structure that contains details about a system specification for a target system. It includes a list of required component types, the corresponding architecture instance and the critical use case instance, etc.

6.3.2 URDS

The tasks of the URDS are to provide an active distributed component management for the USGI in the UniFrame. It attempts to actively discover components and registers them with the Headhunters in the URDS. An important advantage of having such kind of service is that the concrete components are not coded in the *System Generator*, thus, adding and removing a concrete component does not impact the *System Generator*. Chapter 3 provides a brief overview of the URDS. For details of the URDS and the related algorithms, see [SIR02]. The following algorithm (ALGORITHM_1) provides the process for a *URDS Proxy* to interface with URDS service. It interfaces with *QueryManager*, a service component in the URDS. The *URDS Proxy* is responsible for passing the queries for concrete components from the *System Generator* to the URDS, processes the results from the URDS before presenting them to the *System Generator*.

ALGORITHM_1 URDS_PROXY_SEARCH_COMPONENTS

IN: *abstractComponent*

OUT: *availableConcreteComponentList*

Generate a *queryBean* with information from *abstractComponent*

resultTable = CALL *queryManager* with *queryBean*

PUT each value in *resultTable* into *availableConcreteComponentList*

RETURN *availableConcreteComponentList*

END ALGORITHM_1 URDS_PROXY_SEARCH_COMPONENTS

6.3.3 Wrapper and Glue Generator

Considering the heterogeneous nature of components, it is conceivable that the software realization of DCS will require an ensemble of components adhering to different models. This requires adapter components to sit between the heterogeneous components to facilitate their cooperation. Thus, the computational aspect of an adapter component indicates the two models for which it provides interoperability. The adapter components achieve interoperability using the principles of wrap and glue technology [LUQ01]. The research work on the adapter is underway at University of Alabama, a collaborator of the UniFrame research. Figure 6.4 shows a simplified model for an adapter. Each adapter component consists of a bridge (adapter core) and two wrapper and glues. Each adapter core provides translation capabilities for a pair of specific component models. Each wrapper and glue takes care of interfacing with a specific component type (abstract component). The *Wrapper and Glue Generator* is responsible for creating necessary wrapper and glues and assembles them with appropriate adapter core. A bridge can be used for generating multiple adapters. The ALGORITHM_2 outlined below shows the basic steps for creating an adapter. More research is underway to incorporate instrumentation code for QoS measurements into the glues.

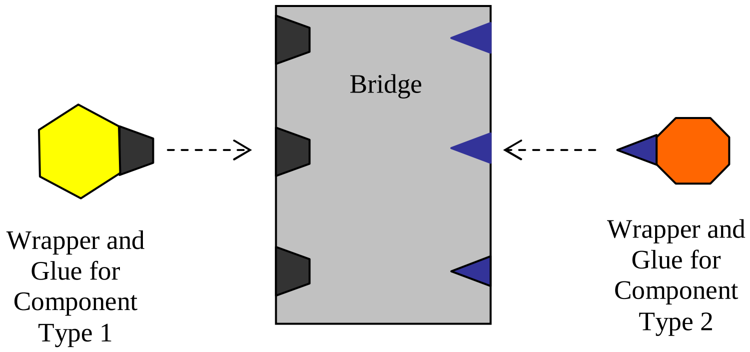


Figure 6.4 Adapter Model

ALGORITHM_2 WGG_GENERATE_WRAPPER_GLUE

IN: *adapterType*

OUT: *adapterComponent* // If *adapterComponent* is NULL,

// it means the adapter cannot be generated.

GET the bridge type from *adapterType*

GET an appropriate bridge for the bridge type

IF the bridge does not exist

RETURN NULL // The adapter can not be generated.

END IF

GET two component types from *adapterType*

Generate appropriate wrapper and glues for the two component types

IF any of the two wrapper and glues can not be generated

RETURN NULL // The adapter can not be generated.

END IF

adapterComponent = Assemble the bridge and two wrapper and glues

RETURN *adapterComponent*

END ALGORITHM_2 WGG_GENERATE_WRAPPER_GLUE

6.3.4 UGDM Knowledge Base (UGDMKB)

The *UGDM Knowledge Base* (UGDMKB) is an important module in the USGI. It stores the UGDM created in the UGDP. The UGDMKB can contain both relational database tables and libraries for a DCS domain. For example, the computing for the system QoS according to the QoS composition rules can be implemented as a library and other models in the UGDM can be represented as relational database tables. The Order Processor and System Generator use the UGDMKB during their activities. There are no special algorithms designed for this module.

6.3.5 UGDMKB Builder Terminal

This module provides a graphical user interface to the UGDMKB builders to access the *UGDMKB Generator*. UGDMKB builders create the UGDMKB through this module. The general algorithm for this module is described in ALGORITHM_3.

```
ALGORITHM_3 UGDMKB_BT_CREATE_UGDMKB
        IN: the domain knowledge of a DCS domain
        OUT: UGDMKB
        CALL ALGORITHM_4 UGDMKBG_CREATE_UGDMKB with the domain
                knowledge of a DCS domain
END ALGORITHM_3 UGDMKB_BT_CREATE_UGDMKB
```

6.3.6 UGDMKB Generator

The *UGDMKB Generator* transforms the UGDM into predefined formats (relational database tables or libraries) to be stored in the UGDMKB. The *UGDMKB Generator* also consists of tools to automate the UGDP to the extent feasible to create the UGDM for a DCS domain. The algorithm provided in ALGORITHM_4 is the overall process for this module at a high conceptual level. Chapter 5 describes the process in detail. Research work is underway to apply the Generic Modeling Environment (GME) [GME] to automate the process.

ALGORITHM_4 UGDMKBG_CREATE_UGDMKB

IN: the domain knowledge of a DCS domain

OUT: UGDMKB

CREATE requirement models (use case model and critical use case model)

DESIGN a layered architecture

CREATE component diagrams

CREATE sequence diagrams

REFINE the critical use case model to the abstract component level

IDENTIFY component interfaces and communication patterns

REFINE the critical use case model to the function/interface level

REFINE the architecture model in disjunctive normal form from component level to function/interface level

MAP the architecture model in disjunctive normal form to the critical use case model (function/interface level)

CREATE the abstract component model

CREATE the QoS composition and decomposition model

GENERATE the UGDM from the artifacts created from the above steps

PUT the UGDM into the UGDMKB

END ALGORITHM_4 UGDMKBG_CREATE_UGDMKB

6.3.7 Application Programmer Terminal

This module interacts with application programmers to fulfill an “order” of a system from a system family. The order can be either in certain predefined formats or in a natural- language-like format. This module passes the order to an *Order Processor* to get the target system specification, and then supplies the system specification to a *System*

Generator to generate the best system that meets the requirement. The general algorithm for this module is described in ALGORITHM_5.

ALGORITHM_5 APT_ORDER_SYSTEM

IN: an order for a system

OUT: *systemBluePrint*

systemSpecification = CALL ALGORITM_6 ORDER_PROCESSOR_ORDER with the order for a system

IF *systemSpecification* is NULL

RETURN NULL // no system can fulfill the requirements

ELSE

systemBluePrint = CALL ALORITHM_7 SG_GENERATE_SYSTEM with *systemSpecification*

RETURN *systemBluePrint*

END IF

END ALGORITHM_5 APT_ORDER_SYSTEM

6.3.8 Order Processor

The *Order Processor* is responsible for determining the target system specification from an “order”. This order can be presented in a predefined format or natural-language like manner. A *Natural Language Processor* (NLP) assists Order Processor in the USGI to process an order in a natural-language-like format. The NLP is based on the theory of Two-Level Grammar (TLG) [BRY02] and natural language specifications [BRY00]. TLG allows queries over the knowledge base, such as a problem space or a solution space, to be stated in a natural-language-like manner. This is consistent with the manner in which the UMM is stated. For details of the TLG and the natural-language-like query processing in the UniFrame, see also [LEE02, LEE02a, BRY02a]. The work on the NLP is underway at University of Alabama at Birmingham, a collaborator of the UniFrame research. The general algorithm for this module is described in ALGORITHM_6.

ALGORITHM_6 ORDER_PROCESSOR_ORDER

IN: an order for a system

OUT: *systemSpecification*

DETERMINE the system architecture instance at the component level according to the mapping from the requirement space to the solution space

DETERMINE the system architecture instance at function/interface level according to the architecture model mapping

DETERMINE the required components according to the system architecture instance at the function/interface level

DETERMINE the critical use case model instance according to the architecture model critical use case model mapping

DETERMINE the expected component QoS according to the QoS decomposition rules

DETERMINE the multiplicity of required components according to the order and the multiplicity model

PUT all information derived above in *systemSpecifiction*

RETURN *systemSpecification*

END ALGORITHM_6 ORDER_PROCESSOR_ORDER

6.3.9 System Generator

The *System Generator* takes a system specification and returns a generated system. The *System Generator* is responsible for system assembly and system validation. The validation includes two phases: static validation (by QoS composition rules) and dynamic validation (by the event grammars with user supplied test cases). The *System Generator* is like an automobile production line. It reads a system specification, acquires the necessary components through the URDS and/or the *Wrapper and Glue Generator*, checks component availability, assembles the components, tests and validates the system, and then releases the product (i.e., an integrated system that satisfies the necessary QoS requirements). Section 6.3.9.1 provides the process for generating a DCS in the UniFrame. The following sections describe various algorithms of the *System Generator*.

Table 6.4 Process for System Generation

Step 1:	Contact the URDS to acquire concrete components for the required abstract components. Check if concrete components are available for all abstract components. If they are available, go to step 2. If concrete components for any abstract components are not available, prompt the application programmer to provide these components. If the application programmer can provide the missing concrete components, go to step 2. If not, abort the process.
Step 2:	Select an appropriate set of concrete components from the available concrete components. If no new set is available, go to step 7.
Step 3:	Check if any adapter is needed for bridging the selected concrete components. If no adapter is needed, go to step 4. Otherwise, contact the URDS to acquire the adapter(s). If the URDS can not find all the needed adapters, contact the *Wrapper and Glue Generator* to generate the wrapper and glues to assemble the adapter(s). If the *Wrapper and Glue Generator* can generate all the needed missing adapters, go to step 4. If not, discard this set of concrete components and go to step 2. Otherwise, this collection forms a potential DCS.
Step 4:	Validate the system QoS statically by the QoS Composition Model. If the system meets the QoS requirements, go to Step 5. If not, discard this set and go to step 2.
Step 5:	Configure the system according to the UGDM.
Step 6:	Validate the system QoS dynamically by the user provided test cases (done by event grammars). If the system meets the QoS requirements, keep the system, otherwise, discard it. Go to step 2.
Step 7:	Select the best system and return the system.

6.3.9.1 Process for System Generation

This section outlines the steps for system generation taken by the *System Generator* in Table 6.4. The algorithms for these steps are described in the following sections. The *System Generator* is the central connection in the USGI. It requires services from all other modules in the framework. The various algorithms reflect its relationship with those modules.

6.3.9.2 Algorithm for Generating a System

ALGORITHM_7 outlines the process for generating a system from a system specification. This algorithm applies the rest of the algorithms designed for the System Generator. It returns the best system in terms of the system QoS. The notion of best system can be defined differently under different circumstances, For example, the best system can be defined as the one with the best system QoS, or the one that has the closest system QoS to the system QoS requirements.

ALGORITHM_7 SG_GENERATE_SYSTEM

IN: *systemSpecification*

OUT: *systemBluePrint*

availableConcreteComponentTable = CALL ALGORITHM_8

SG_ACQUIRE_CONCRETE_COMPONENTS

with *systemSpecification*

IF *availableConcreteComponentTable* is NULL

RETURN NULL //Abort the process.

// The system specification cannot be fulfilled.

ELSE

Generate *selectedConcreteComponentTableQueue* from

availableConcreteComponentTable and *systemSpecification*

WHILE *selectedConcreteComponentTableQueue* is NOT empty

CREATE a *systemBluePrint* for the possible target system instance

selectedConcreteComponentTable = Remove one table from
selectedConcreteComponentListQueue
ADD *selectedConcreteComponentTable* to *systemBluePrint*
adapterTypeList = CALL ALGORITHM_9
SG_DETERMINE_ADAPTER_TYPES with
selectedConcreteComponentTable
and *systemSpecification*
IF *adapterTypeList* is NULL //Fail to determine required adapters
Continue to the END WHILE // Discard the combination
ELSE
ADD *adapterTypeList* to *systemBluePrint*
IF *adapterTypeList* is NOT empty
availableAdapterTable = CALL ALGORITHM_10
SG_ACQUIRE_ADAPTERS
with *adapterTypeList*
IF *availableAdapterTable* is NULL
Continue to the END WHILE // Discard the
// combination
END IF
END IF
staticSystemQoS = CALL ALGORITHM_11
SG_GET_STATIC_SYSTEM_QOS
IF *staticSystemQoS* does not meet *expectedSystemQoS* in
systemSpecification
Continue to the END WHILE // Discard the
// combination
ELSE
ADD *staticSystemQoS* to *systemBluePrint*
CALL ALGORITHM_12
SG_ASSEMBLE_SYSTEM

IF system assembly failed
Continue to the END WHILE // Discard the
// combination
END IF
dynamicSystemQoS = CALL ALGORITHM_13
SG_GET_DYNAMIC _SYSTEM_QOS
IF *dynamicSystemQoS* does not meet
expectedSystemQoS in *systemSpecification*
Continue to the END WHILE // Discard the
// combination
ELSE
ADD *dynamicSystemQoS* to *systemBluePrint*
Put *systemBluePrint* for this target system
instance in *systemBluePrintList*
END IF
END IF
END IF
END WHILE
END IF
SORT *systemBluePrintList* by *dynamicSystemQoS*
systemBluePrint = GET *systemBlueprint* with the best *dynamicSystemQoS* from
the sorted *systemBluePrintList*
RETURN *systemBluePrint*
END ALGORITHM_7 SG_GENERATE_SYSTEM

<u>6.3.9.3 Algorithm for Acquiring Concrete Components</u>

ALGORITHM_8 outlines the process for acquiring the concrete components via the URDS for the required abstract components in a system specification. The request is passed from the System Generator to the URDS through the *URDS Proxy*.

ALGORITHM_8 SG_ACQUIRE_CONCRETE_COMPONENTS

IN: *systemSpecification*

OUT: *availableConcreteComponentTable*

// If *availableConcreteComponentTable* is NULL, it means some abstract
// components do not have available concrete components. Thus, the
// system specification cannot be fulfilled.

requiredAbstractComponentList = GET the list of required abstract components from *systemSpecification*

FOREACH abstract component in *requiredAbstractComponentList*

availableConcreteComponentList = GET a list of available concrete components for the abstract component from the URDS.

IF *availableConcreteComponentList* is empty

availableConcreteComponentList = GET available concrete components for the abstract component from the Application Programmer.

END IF

IF *availableConcreteComponentList* is empty

Return NULL //Abort the process. The system specification can not be fulfilled.

ELSE

PUT the abstract component name and *availableConcreteComponentList* in *availableConcreteComponentTable*

END IF

END FOREACH

RETURN *availableConcreteComponentTable*

END ALGORITHM_8 SG_ACQUIRE_CONCRETE_COMPONENTS

6.3.9.4 Algorithm for Determining Adapter Types

ALGORITHM_9 outlines the process of determining the adapter types for a selected combination of concrete components that forms a possible target system instance.

ALGORITHM_9 SG_DETERMINE_ADAPTER_TYPES
    IN: *selectedConcreteComponentTable, systemSpecification*
    OUT: *adapterTypeList* // 1) If *adapterTypeList* is empty, it means no
        // need for adapters. 2) If *adapterTypeList* is NULL, it
        // means some adapter type cannot be determined; thus, the
        // system specification cannot be fulfilled.
    GET the domain name from the *systemSpecification*
    bridgeTable = GET the bridge table from the UGDMKB for the domain
    componentInteractionTable = GET the abstract component interaction table from
        the UGDMKB
    FOREACH concrete component (C1) in *selectedConcreteComponentTable*
        postprocessingCollaboratorList = GET the corresponding list of post-
            processing collaborator types from *componentInteractionTable* for
            the type of the concrete component
        IF *postprocessingCollaboratorList* is NOT empty
            FOREACH abstract component in *postprocessingCollaboratorList*
                selectedConcreteComponentList = GET the selected
                    concrete component list for the abstract component
                    from *selectedConcreteComponentTable*
                FOREACH concrete component (C2) in
                    selectedConcreteComponentList
                    IF C1 and C2 are of different component models
                        GET the bridge type from *bridgeTable*
                        IF the bridge type exists

PUT the bridge type, two abstract components and two component models (for C1 and C2) in *adapterTypeList*

ELSE

RETURN NULL //Abort the task.

//The requirement cannot be

// fulfilled.

END IF

END IF

END FOREACH

END FOREACH

END IF

END FOREACH

RETURN *adapterTypeList*

END ALGORITHM_9 SG_DETERMINE_ADAPTER_TYPES

6.3.9.5 Algorithm for Acquiring Adapters

ALGORITHM_10 outlines the process for the *System Generator* to acquire adapters from the URDS via the *URDS Proxy*. If no adapter is found, the *System Generator* sends the request to the *Wrapper and Glue Generator*.

ALGORITHM_10 SG_ACQUIRE_ADAPTERS

IN: *adapterTypeList*

OUT: *availableAdapterTable*

// If *availableAdapterTable* is NULL, it means some adapter types do not

// have available instances; thus, the system specification cannot

// be fulfilled.

FOREACH adapter in *adapterTypeList*

availableAdapterList = GET a list of adapter instances for the adapter type from the URDS.

IF *availableAdaperList* is empty

availableAdapterList = GET the adapter instances from the Wrapper and Glue Generator

END IF

IF *availableAdapterList* is empty

RETURN NULL //Abort the process, the task can not be fulfilled.

ELSE

PUT the name of the adapter type and *availableAdapterList* in *availableAdapterTable*

END IF

END FOREACH

RETURN *availableAdapterTable*

END ALGORITHM_10 SG_ACQUIRE_ADAPTERS

6.3.9.6 Algorithm for Getting Static System QoS

ALGORITHM_11 outlines the process for getting the static system QoS from the QoS Composition Model, which is implemented as a library. The static system QoS is predicted from the component QoS advertised for the concrete components by the component developers. These component QoS are documented in the UMM specifications when the concrete components are deployed over the network.

ALGORITHM_11 SG_GET_STATIC_SYSTEM_QOS

IN: *selectedConcreteComponentList*, *systemSpecification*

OUT: *staticSystemQoS*

GET *qosCompositionModel* for the domain from UGDMKB

// The model is implemented as a library

staticSystemQoS = CALL *qosCompositionModel* with *selectedConcreteComponentList* and *systemSpecification*

RETURN *staticSystemQoS*

END ALGORITHM_11 SG_ GET_STATIC_SYSTEM_QOS

6.3.9.7 Algorithm for Assembling a System

ALGORITHM_12 outlines the process for assembling a system from the selected concrete components and possible necessary adapters. The configuration knowledge used in system assembling includes the component interaction model and the component-level multiplicity model. Domain dependent configuration knowledge can also be defined when necessary.

ALGORITHM_12 SG_ASSEMBLE_SYSTEM

IN: *systemBluePrint*

OUT: boolean // 1) true: system is assembled successfully

// 2) false: assembly failed due to some reason,

// such as network errors, etc.

requiredAbstractComponentList = GET the list of required abstract components from the *systemBluePrint*

selectedConcreteComponentTable = GET the table from *systemBluePrint*

componentInteractionTable = GET the abstract component interaction table from the UGDMKB

requiredAdapterList = GET the list of required adapters from *systemBluePrint*

foundAdapterTable = GET the found adapters from *systemBluePrint*

multiplicityModel = GET the component-level multiplicity model from the UGDMKB

FOREACH concrete component in *selectedConcreteComponentTable*

LOCK the component for assembly (exclusively for a system assembler)

END FOREACH

//configure the system according to the component-level multiplicity model

FOREACH abstract component (A1) in *requiredAbstractComponentList*

postProcessingCollaboratorList = GET the list of the post-processing
collaborators for the abstract component from
componentInteractionTable

IF *postProcessingCollaboratorList* is NULL

Continue to the END FOREACH

END IF

selectedConcreteComponentList (initiator) = GET the list of the selected
concrete components for the abstract component A1 from
selectedConcreteComponentTable

FOREACH abstract component (A2) in *postProcessingCollaboratorList*

selectedConcreteComponentList (responder) = GET the list of the
selected concrete components for the abstract component
A2 from *selectedConcreteComponentTable*

IF the multiplicity of A1 to A2 is one to one

FOREACH concrete component (C1) in
selectedConcreteComponentList (initiator)

GET a concrete component (C2) from
selectedConcreteComponentList (responder)

GET component ID (ID1) from C1

IF C1 and C2 are of the same technology

GET component ID (ID2) from C2

ELSE

GET *adapterType* for C1 and C2 from
requiredAdapterTypeList

GET the adapter from *foundAdapterTable*
for adapterType

GET component ID (ID2) from the adapter

END IF

GET the handle to C1 by ID1

CONFIGURE C1 with ID2

END FOREACH

ELSE IF the multiplicity of A1 to A2 is one to many

FOREACH concrete component (C1) in *selectedConcreteComponentList (initiator)*

FOREACH concrete component (C2) in *selectedConcreteComponentList (responder)*

GET component ID (ID1) from C1

IF C1 and C2 are of the same technology

GET component ID (ID2) from C2

ELSE

GET *adapterType* for C1 and C2 from *requiredAdapterTypeList*

GET the adapter from *foundAdapterTable* for *adapterType*

GET component ID (ID2) from the adapter

END IF

GET the handle to C1 by ID1

CONFIGURE C1 with ID2

END FOREACH

END FOREACH

ELSE IF the multiplicity of A2 to A1 is one to many

FOREACH concrete component (C2) in *selectedConcreteComponentList (responder)*

FOREACH concrete component (C1) in *selectedConcreteComponentList (initiator)*

GET component ID (ID1) from C1

IF C1 and C2 are of the same technology

GET component ID (ID2) from C2

```
ELSE
    GET adapterType for C1 and C2
        from requiredAdapterTypeList
    GET the adapter from
        foundAdapterTable for
        adapterType
    GET component ID (ID2) from the
        adapter
END IF
GET the handle to C1 by ID1
CONFIGURE C1 with ID2
```

END FOREACH

END FOREACH

END IF // other multiplicity situation also needs to be handled

// those listed are some most common situations

END FOREACH

END FOREACH

CONFIGURE other domain dependent configuration knowledge if necessary

END ALGORITHM_12 SG_ASSEMBLE_SYSTEM

6.3.9.8 Algorithm for Getting Dynamic System QoS

ALGORITHM_13 outlines the process for getting dynamic system QoS through the event grammars model (the system behavior model), which is being developed at New Mexico State University, a collaborator of the UniFrame research.

ALGORITHM_13 SG_ GET_DYNAMIC_SYSTEM_QOS

IN: *systemBluePrint*

OUT: *dynamicSystemQoS*

GET handler to *evevenGrammarModel* for the domain.

dynamicSystemQoS

= CALL *eventGrammarModel* with the customer supplied test cases

RETURN *dynamicSystemQoS*

End ALGORITHM_13 SG_ GET_DYNAMIC_SYSTEM_QOS

This chapter presents in detail the high level concepts of the USGI in the USGPF. The description covers the architecture, the workflow modeling of the system and the algorithms for each module in the framework. In the next chapter, a prototype design and implementation with multi-tier architecture for the USGI is described. The banking domain example developed in Chapter 4 and Chapter 5 serve as the example to demonstrate the prototype.

7. THE USGI PROTOTYPE DESIGN AND IMPLEMENTATION

Chapter 6 describes the USGI at the conceptual level. The architecture, workflow modeling and algorithms presented in Chapter 6 do not adhere to any specific implementational technology. The USGI can be realized in several different technologies. In this chapter, the details of a prototype design and implementation of the USGI using Java is presented. The prototype serves to demonstrate the feasibility of the proposed USGPF in this thesis and allows experimentation with it.

7.1 Technology

This section describes the J2EE™ technology [SM01, SM02, SM02a] that is the model for designing and implementing the prototype of the USGI. J2EE™ technology provides a component-based approach to the design, development, assembly, and deployment of enterprise applications. The J2EE™ platform offers a multi-tiered distributed application model, the ability to reuse components, integrated Extensible Markup Language (XML)-based data interchange, a unified security model, and flexible transaction control.

7.1.1 J2EE™ Application Model

A J2EE™ application uses a multi-tiered distributed application model. In this model, the application logic is divided into components according to functions. The various application components that make up a J2EE™ application are installed on different machines. The installation of the components depends on the tier to which the application component belongs in the multi-tiered J2EE™ environment. The multi-tiered architecture is an extension of the traditional two-tier client-server model [SM02a]. In a four-tier architecture, the client is replaced by a web browser and HTML pages powered by servlet/JavaServer Pages^{TM} technology hosted on a web server. A multithreaded application server sits between the web sever and a backend database. Figure 7.1 shows the four-tier architecture of J2EE^{TM} applications [SM02].

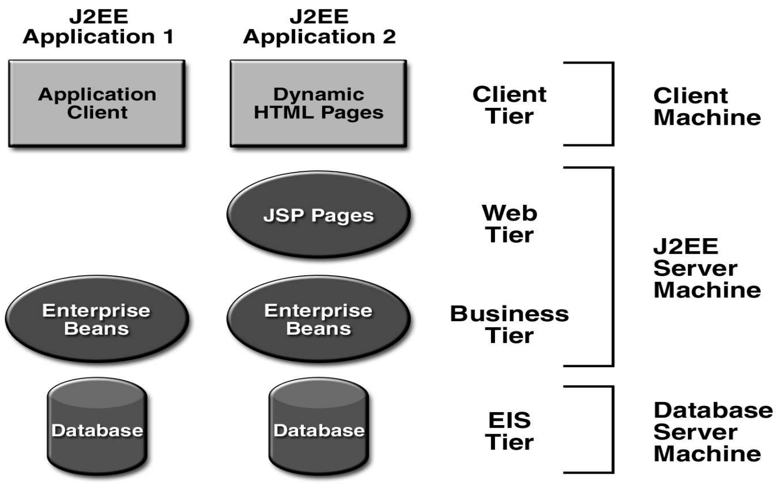


Figure 7.1 Multi-tier Architecture of J2EE^{TM} Applications (from [SM02])

7.1.2 J2EE^{TM} Components

J2EE^{TM} applications are made up of J2EE^{TM} components [SM02, SM02a]. A J2EE^{TM} component is a self-contained functional software unit that is assembled into a J2EE^{TM} application with its related classes and files, and it communicates with other components. The J2EE^{TM} specification defines the following J2EE^{TM} components: *Client Components*, *Web Components* and *Business Components*.

7.1.2.1 Client Components

Client Components run on client machines. The *Client Components* include *Web Clients*, *Applets* and *Application Clients*.

A *Web Client* consists of two parts: dynamic Web pages containing various types of markup language (e.g., HTML and XML), which are generated by *Web Components* running in the Web tier, and a Web browser, which renders the pages received from the server. A Web client is sometimes called a *thin client*. Thin clients usually do not do things like query databases, execute complex business rules, or connect to legacy applications.

A Web page received from the Web tier can include an embedded *Applet*. An *Applet* is a small client application written in the Java programming language that executes in the Java virtual machine installed in the Web browser. However, client systems will likely need a Java Plug-in and possibly a security policy file in order for the applet to successfully execute in the Web browser.

An *Application Client* provides a way for users to handle tasks that require a richer user interface than can be provided by a markup language. It typically has a graphical user interface (GUI) created from Swing or Abstract Window Toolkit (AWT) APIs.

7.1.2.2 Web Components

J2EE™ *Web Components* can be either servlets or Java Server Pages (JSP). Servlets are Java programming language classes that dynamically process requests and construct responses. JSP pages are text-based documents that execute as servlets but allow a more natural approach to creating static content. Static HTML pages and applets are bundled with *Web Components* during application assembly, but are not considered *Web Components* by the J2EE™ specification. Server-side utility classes can also be bundled with *Web Components* and, like HTML pages, are not considered *Web Components*. Like the client tier, the Web tier might include a JavaBeans component [STE00], which is discussed in Section 7.1.2.4, to manage the user input and send that input to enterprise beans running in the business tier for processing.

7.1.2.3 Business Components

Business Components are Enterprise JavaBeans™ (EJB™) components (enterprise beans), which are deployed on application servers and form the business tier. The business components provide the business logic that solves or meets the needs of a particular business domain such as banking, retail, or finance. The heavyweight operations in clients in the traditional client-server model are off-loaded to enterprise beans executing on the application server where they can leverage the security, speed, services, and reliability of J2EE™ server-side technologies.

7.1.2.4 JavaBeans Component

The server and client tiers might also include components (JavaBeans components) based on the JavaBeans Component Architecture [SM03a] to manage the data flow between an application client or applet and components running on the J2EE™ server or between server components and a database. JavaBeans components are not considered J2EE™ components by the J2EE™ specification. JavaBeans components are reusable software components that are written in the Java programming language. JavaBeans components have instance variables and *get* and *set* methods for accessing the data in the instance variables. JavaBeans components used in this way are typically simple in design and implementation, but should conform to the naming and design conventions outlined in the JavaBeans component architecture.

7.1.3 Service Technologies

The J2EE™ platform [SM02] service technologies allow applications to access a variety of services. The prominent service technologies supported are JDBC™ API [SM03b] which provides access to databases, Java Transaction API (JTA) [SM03c] for transaction processing, Java Naming and Directory Interface™ (JNDI) [SM03d] which provides access to naming and directory services, J2EE™ Connector Architecture [SM03e] which supports access to enterprise information systems, and Java API for

XML Processing (JAXP) [SM03f] which enables applications to parse and transform XML documents independent of a particular XML processing implementation**.** The service technologies used in the prototype are described below.

7.1.3.1 JDBC^{TM} API 2.0

The JDBC^{TM} API provides methods to invoke SQL commands from Java programming language methods. The JDBC API has two parts: an application-level interface used by the application components to access a database, and a service provider interface to attach a JDBC driver to the J2EE^{TM} platform.

7.1.3.2 Java API for XML Processing 1.1

XML is a language for representing text-based data so the data can be read and used by any program or any tool. Programs and tools can generate XML documents that other programs and tools can read and use. Java API for XML Processing (JAXP) supports processing of XML documents using DOM, SAX, and XSLT parsers. Depending on the needs of the application, developers have the flexibility to swap between XML processors (such as high performance vs. memory conservative parsers) without making application code changes.

7.1.4 Communication Technologies

Communication technologies provide mechanisms for communication between clients and servers and between collaborating objects hosted by different servers. Some of the communications technologies supported by the J2EE^{TM} Platform [SM01] include Transport Control Protocol over Internet Protocol (TCP/IP), Hypertext Transfer Protocol HTTP, Secure Socket Layer SSL, Java Remote Method Protocol (JRMP), Java IDL, Remote Method Invocation over Internet Inter ORB Protocol (RMI-IIOP), Java Message Service (JMS), JavaMail and Java Activation Framework. The prototype uses the *HTTP*

1.0 Protocol for communication between the browser-based clients and server side components. The inter-component communication on the server side is achieved through Java Remote Method Invocation (RMI). The communication techonologies used in the prototype are described below.

7.1.4.1 HTTP 1.0 Protocol

The Hypertext Transfer Protocol (HTTP) [WWW03] HTTP has been in use by the World-Wide Web global information initiative since 1990. It is an application-level, generic stateless protocol for distributed, collaborative, hypermedia information systems. A feature of HTTP is the typing and negotiation of data representation, allowing systems to be built independently of the data being transferred.

7.1.4.2 Java Remote Method Invocation (RMI)

Java Remote Method Invocation (RMI) [SM03g] is a set of APIs in the Java programming language that enables developers to build distributed applications. RMI uses Java language interfaces to define remote objects, and it combines Java serialization technology and the Java Remote Method Protocol (JRMP) for performing remote method invocations. JRMP is a proprietary stream-base protocol on top of the TCP/IP.

7.2 USGI Prototype Design

The USGI prototype is designed as a multi-tiered distributed application based on the J2EE^{TM} model. The USGI functionality is partitioned into modules, and these modules are decomposed into specific objects to represent the behavior and data of the application. The prototype adopts the Model-View-Controller (MVC) architecture. The MVC architecture [GAM95, YOU95] can be described as: "The *Model* represents the application data and the rules that govern access and modification of this data. The *View* renders the contents of a model. It accesses data from the model and defines how that data should be presented. The *Controller* defines application behavior; it translates user gestures into actions to be performed by the model". The design of the USGI is shown in Figure 7.2.

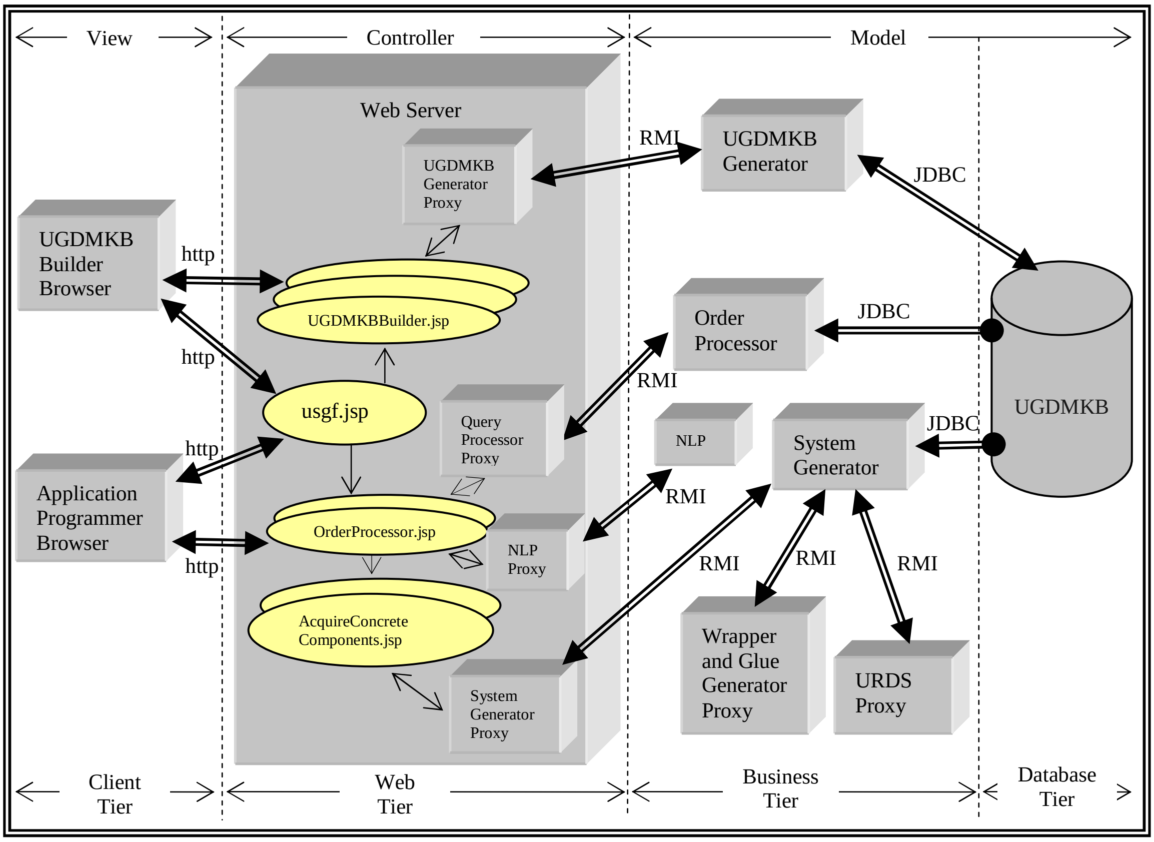


Figure 7.2 USGI Prototype Design

The *View* of the USGI consists of browsers, which provides the interfaces for the users (UGDMKB Builders and Application Programmers/System Assemblers/System Integrators) to interact with the system. The *View* of the USGI forms the *Client Tier* of the USGI architecture.

The *Model* of the USGI consists of the *Business Tier* and the *Database Tier* of the USGI architecture. The *Business Tier* includes the following modules of the USGI: UGDMKB Generator, Order Processor, System Generator, Wrapper and Glue Generator

Proxy, URDS Proxy, and Natural Language Processor (NLP), which is actually a proxy to the NLP implemented in C++ by the University of Alabama, a collaborator of the UniFrame research. The *Database Tier* includes the relational database tables of the UGDMKB. The library of the UGDMKB is not shown in the figure as an individual module. The library is used directly by the System Generator.

The *Controller* of the USGI consists of two parts: proxy classes and JSP pages. The proxy classes mediate information exchanges between the JSP Pages and the modules in the *Business Tier*. The JSP pages receive inputs from and render results to users via the browsers in the *Client Tier*.

7.3 USGI Prototype Implementation

This section describes the implementation of the USGI prototype using Java in detail. Before the description of the implementation, the outline of the platform and the environment, and the communication infrastructure for the implementation are described.

7.3.1 Platform and Environment

In the prototype created for this thesis, the algorithms outlined for the various modules in Chapter 6 are implemented using the Java^{TM} 2 Platform, Standard Edition (J2SE) version 1.4.0 [SM03]. The service components in the business tier are implemented as Java-RMI based services. The UGDMKB is a database-oriented implementation based on Oracle, version 8.1.7 [ORA03]. The web-based components (JSPs), which service client interactions, are housed in the Tomcat 3.3a Servlet/JSP Container [APA03a].

7.3.2 Communication Infrastructure

The communication between proxy classes in the *Web Tier* and the *Business Components* and the communication among the *Business Components* are based on Java

RMI. The connections to the databases are established using the JDBC APIs. Interactions between the clients (users) and the *Web Components* are based on the HTTP protocol.

7.3.3 Implementation Details

This section organizes the details of the USGI implementation according to the four-tier architecture. The description is presented in the following order: *Client Tier*, *Web Tier*, *Business Tier* and *Database Tier*.

7.3.3.1 Client Tier

The client tier consists of the *Client Components*, which are web browsers in the USGI. These *Client Components* provide the views (i.e., the interfaces) for the USGI, through which the users (UGDMKB Builders and Application Programmers) can interact with the system. The JSP and JavaBeans work together to implement the views. The JSP pages dynamically generate html pages. JavaBeans encapsulate information exchanges between *Client Components*, *Web Components* and *Business Components*. The following JSP pages provide views to users: *USGI.jsp*, *OrderWithoutNLP.jsp*, *OrderWithNLP.jsp*, *Order.jsp*, *AvailableConcreteComponents.jsp*, *SelectedConcreteComponents.jsp*, *DetermineAdapterTypes.jsp*, *AcquireAdapters.jsp*, *DynamicComponentQoS.jsp*, *StaticSystemValidation.jsp*, *DynamicSystemValidation.jsp*, *ComponentDescription.jsp*, and *UGDMKBGeneration.jsp*. These JSP pages are part of the *Web Components*, which forms the *Controller* in the MVC architecture. Not all the JSP pages in the *Web Tier* provide views to the users. The *Web Components* are explained in the next section.

7.3.3.1.1 View Provided by *usgi.jsp*

Figure 7.3 illustrates the view provided by *usgi.jsp*. This view shows the main demonstrations available in the prototype: System Generation without NLP, System

Generation with NLP and UGDMKB Generation. This view also provides choices for running the USGI under different simulation modes.

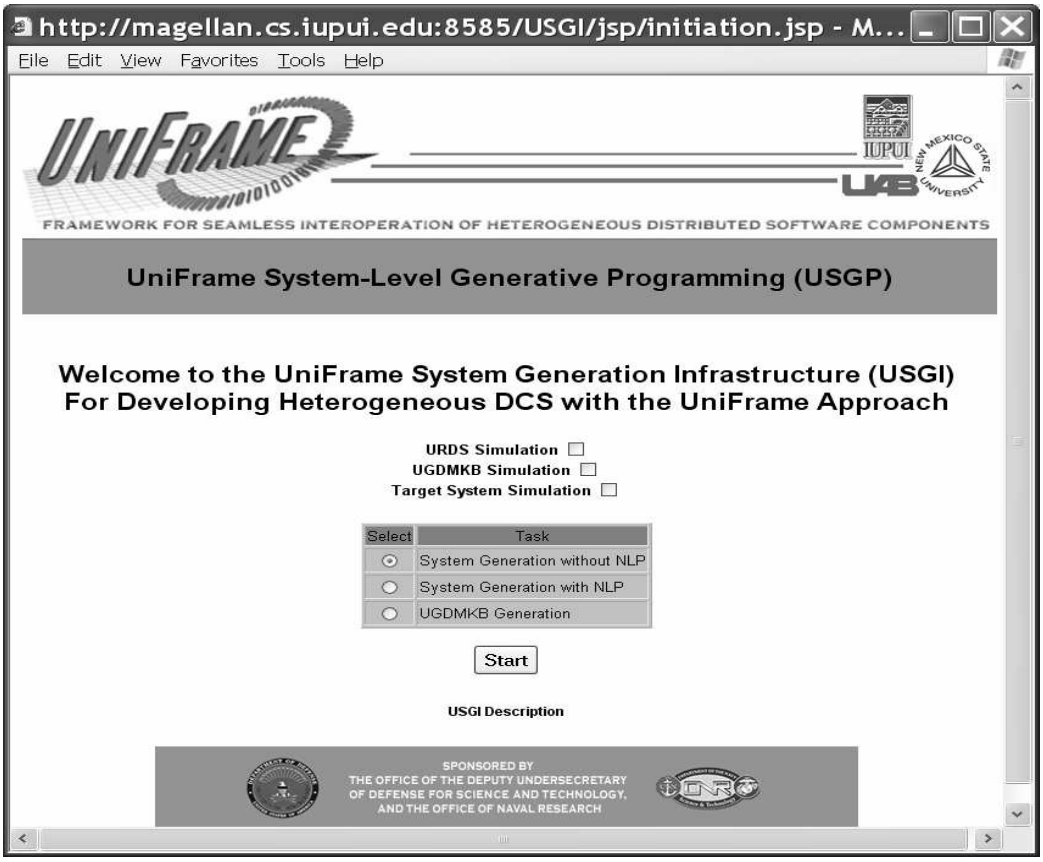


Figure 7.3 The View Provided by *usgi.jsp*

System Generation without NLP demonstrates the ordering of a system without natural language query processing support. On the other hand, System Generation with NLP demonstrates the ordering of a system with natural language query processing support. UGDMKB Generation demonstrates the processing of the UGDM models in the XML format into the Oracle database by using the XML parsers implemented with the Xerces Java Parser from Apache [APA03]. These parsers are described in Section 7.3.3.3.1.

The simulation modes include *URDS Simulation*, *UGDMKB Simulation* and *Target System Simulation*. The *URDS Simulation* simulates the URDS functionality of searching for concrete components of the banking domain by local data structures. The *UGDMKB Simulation* also simulates the relational database tables of the UGDMKB for the banking domain by local data structures. Under the *Target System Simulation*, the dynamic component QoS testing, the system assembly and the dynamic system QoS validation are simulated without running the bank components. These modes provide a convenient way to demonstrate the USGI without setting up the whole system.

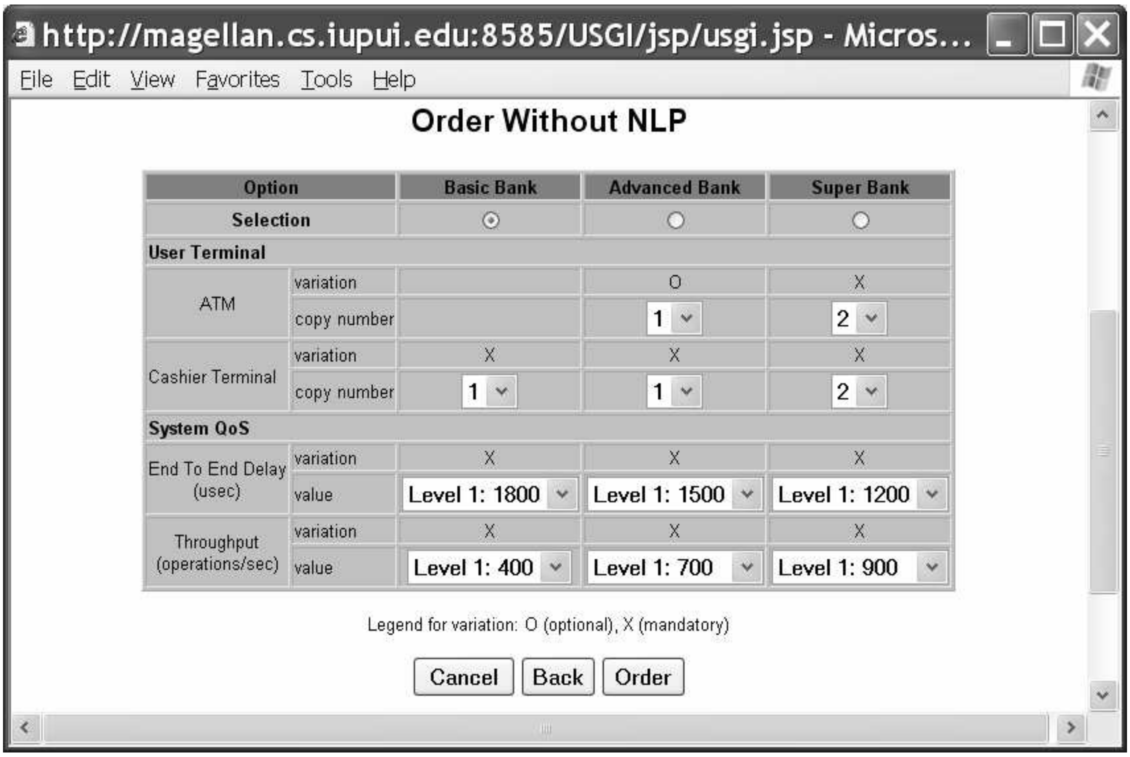


Figure 7.4 View Provided by *OrderWithoutNLP.jsp*

7.3.3.1.2 View Provided by *OrderWithoutNLP.jsp*

Figure 7.4 illustrates the view provided by *OrderWithoutNLP.jsp*. This view is the user interface to order a system without natural language processing support. The ordering language is implemented as a table. This is an example of the implementation of a tabular DSL for ordering a system from the banking domain developed in Chapter 5. The view allows application developers to select different options of bank system types: *Basic Bank*, *Advanced Bank* and *Super Bank*, which differ from each other as shown by the parameters in the figure: number of user terminals (both *ATM* and *Cashier Terminal*) and system QoS requirements (*end to end delay* and *throughput*). It allows specifying different parameters to order a bank system from a selected bank system type.

7.3.3.1.3 View Provided by *OrderWithNLP.jsp*

Figure 7.5 illustrates the view provided by *OrderWithNLP.jsp*. This view is the user interface to order a system with natural language query processing support. Ordering requirements are input as natural-language-like style in the provided text area. An example is shown in Figure 7.5.

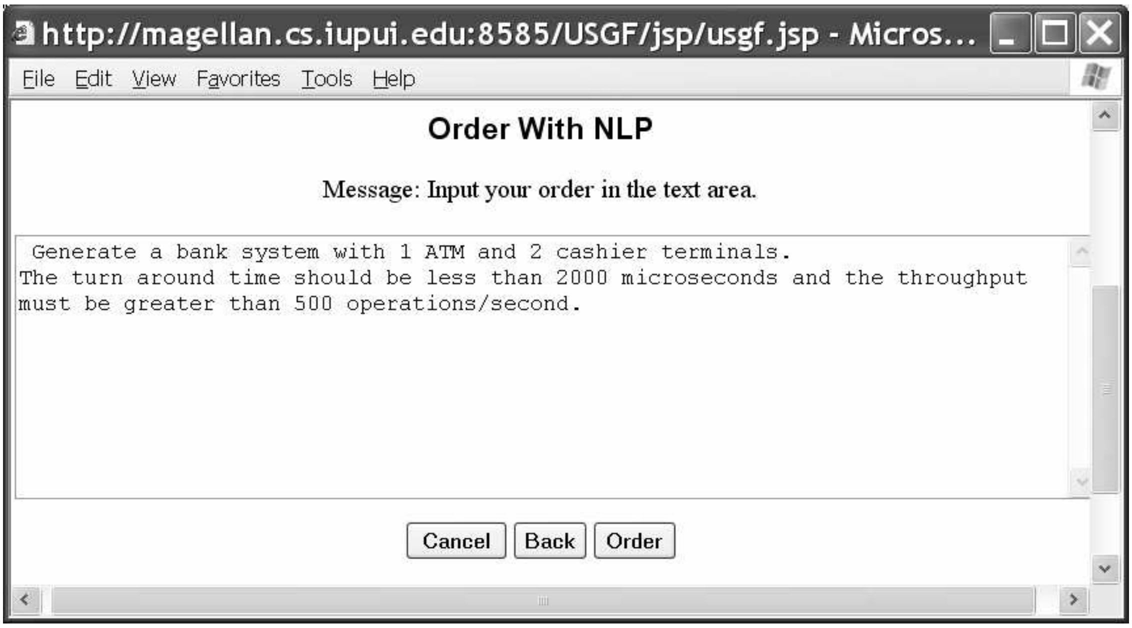


Figure 7.5 View Provided by *OrderWithNLP.jsp*

7.3.3.1.4 View Provided by *Order.jsp*

Figure 7.6 illustrates the view provided by *Order.jsp*. In this view the system specification that meets the ordering requirement is displayed. The system specification in the view includes *System Name*, *Architecture ID* (both at the component level and at the interface level), *Expected System QoS*, *Required Abstract Components* and their *Expected Component QoS* which is derived through QoS Decomposition Model.

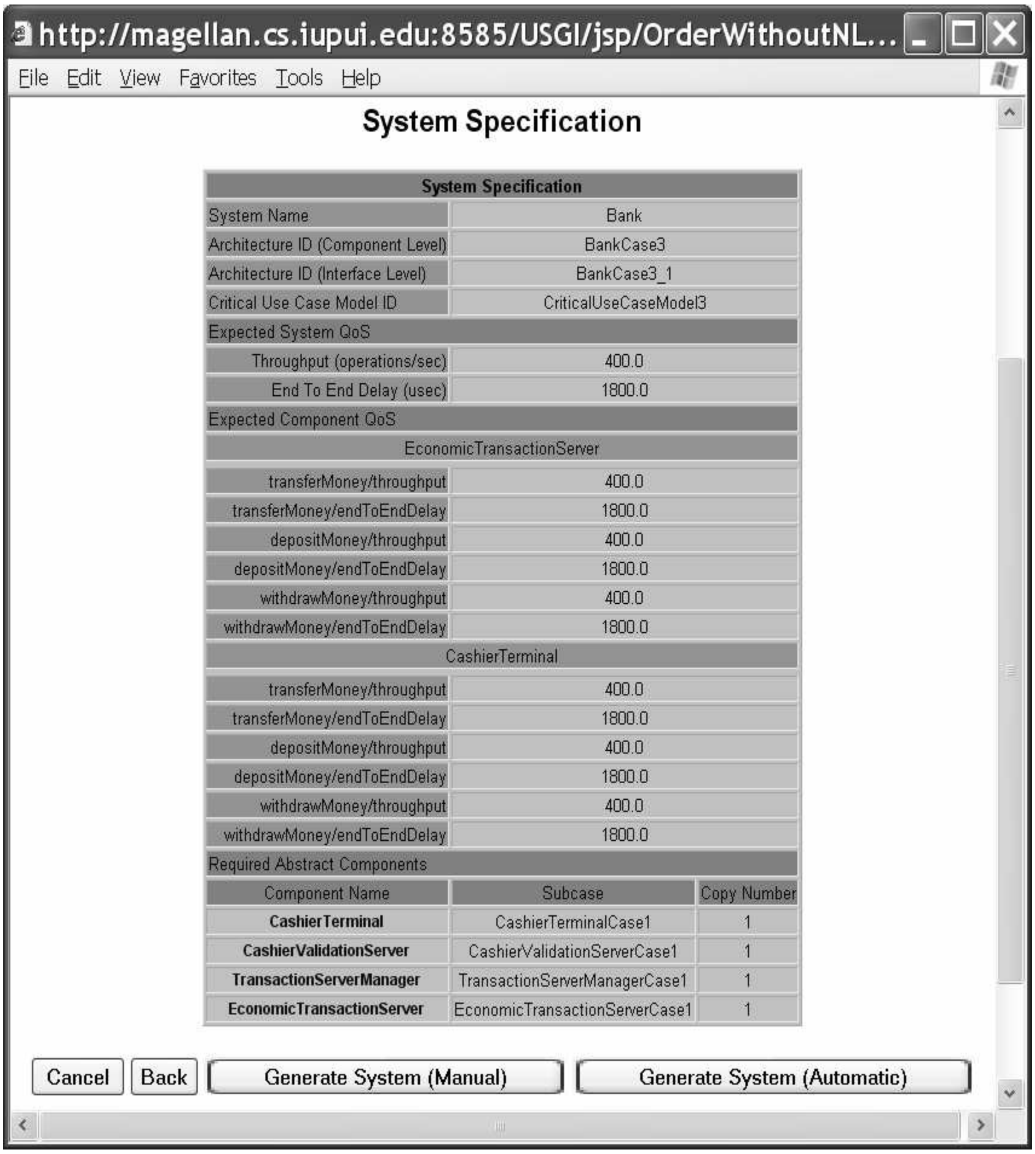


Figure 7.6 View Provided by *Order.jsp*

This view also illustrates the two system generation choices provided by the USGI: *Generate System (Manual)* and *Generate System (Automatic)*. In the choice of *Generate System (Manual)*, the system allows users to interact with and make decisions during the system generation process. In the choice of *Generate System (Automatic)*, the system will go through all the possibilities and return the best system to users. The best system can be defined as the one with the best system QoS, or as the one with the closest system QoS to the ordering requirements. In the choice of *Generate System (Manual)*, the users have the choice to decide which system is the best to their needs.

7.3.3.1.4 View Provided by *AvailableConcreteComponents.jsp*

Figure 7.7 illustrates the view provided by *AvailableConcreteComponents.jsp*. This view shows whether concrete components have been found for the required abstract components or not. If concrete components for a required abstract component have been found, a click on the link under the column of *Searching Result* brings a list of concrete components that were found by the URDS. If no concrete components have been found for a particular abstract component, then the entry of the correspondent cell under the column of *Searching Result* is “NOT FOUND”. This view also displays a message asking users to select a set of concrete components to assemble an integrated system.

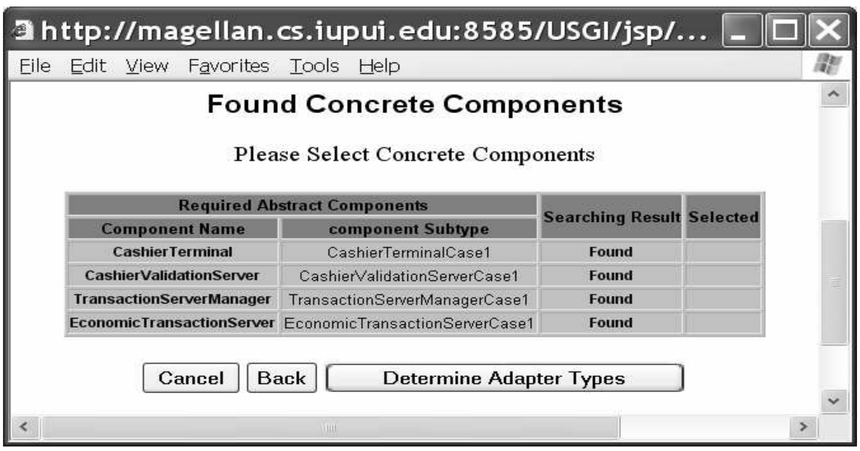


Figure 7.7 View Provided by *AvailableConcreteComponents.jsp*

7.3.3.1.5 View Provided by *SelectConcreteComponents.jsp*

Figure 7.8 illustrates the view provided by *SelectConcreteComponents.jsp*. This view displays a list of found concrete components for a specific abstract component. Users can check the checkbox to select components. A click on the link under the column of *ComponentID* brings the UMM specification for the corresponding concrete component. The links under the column of *Dynamic QoS Tesing* direct users to the page to test the QoS of the components.

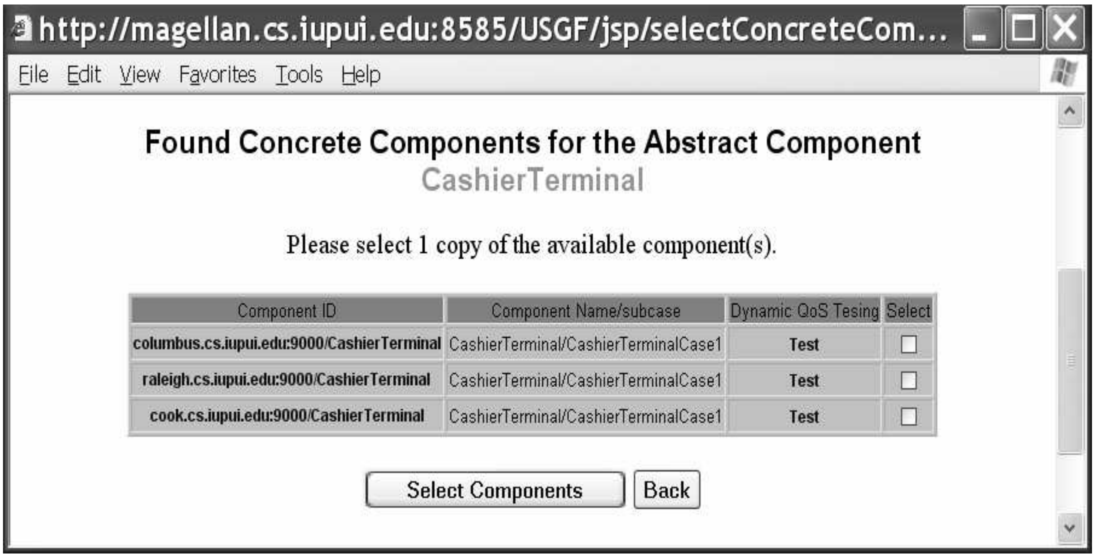


Figure 7.8 View Provided by *SelectConcreteComponents.jsp*

7.3.3.1.6 View Provided by *DetermineAdapterTypes.jsp*

Figure 7.9 illustrates the view provided by *DetermineAdapterTypes.jsp*. This view displays the required adapter types based on the set of selected concrete components. Each row of the table in the view indicates one required adapter type, which is described by three columns: *Bridge Type*, *Preprocessing Component* and *Postprocessing Component*. The *Bridge Type* indicates the two component models that the adapter is capable of bridging. The *Preprocessing Component* initiates the interaction and the

Postprocessing Component responds in the interaction. The interactions between these two heterogeneous components are mediated by the adapter.

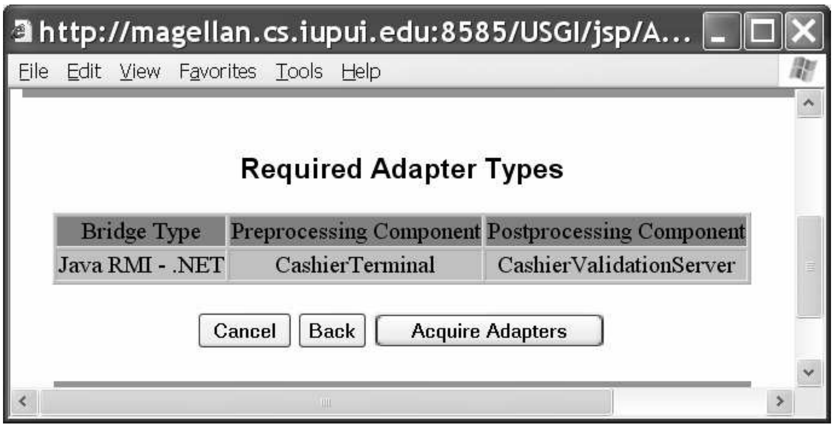


Figure 7.9 View Provided by *DetermineAdapterTypes.jsp*

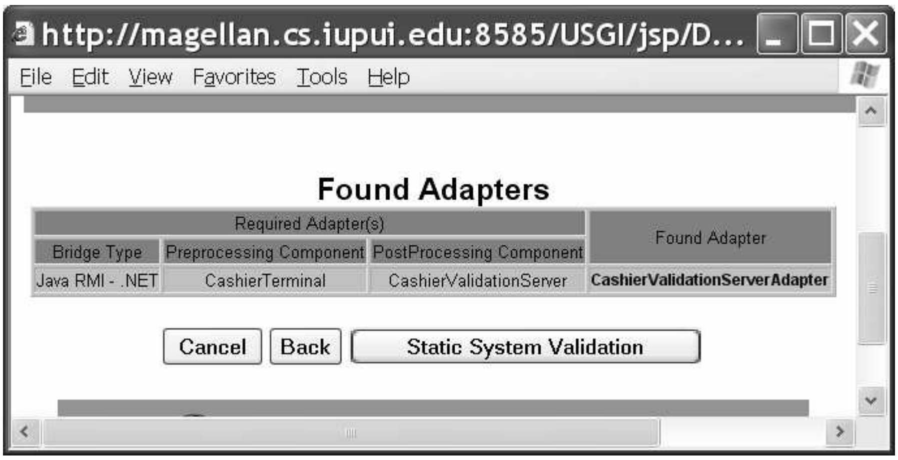


Figure 7.10 View Provided by *AcquireAdapters.jsp*

7.3.3.1.7 View Provided by *AcquireAdapters.jsp*

Figure 7.10 illustrates the view provided by *AcquireAdapters.jsp*. This view displays the adapters found for the required adapter types by the URDS. The adapters themselves are concrete components. The link under the column *Found Adapter* leads to the UMM description for the adapter. In the current prototype, if more than one adapter is found for a required adapter type, the system randomly selects one adapter.

7.3.3.1.8 View Provided by *DynamicComponentQoS.jsp*

Figure 7.11 illustrates the view provided by *DynamicComponentQoS.jsp*. This view displays the results of testing component QoS dynamically in the column called *Dynamic QoS*. The advertised QoS of the component is also displayed. The difference between these two is shown in the column called *Deviation*. The dynamic testing of the component QoS assumes that the component developer provides a test interface for the component and this interface is used to validate the advertised component QoS values. If no such interface is provided, this view displays the message “This component can not be tested. No testing mechanism is available.”

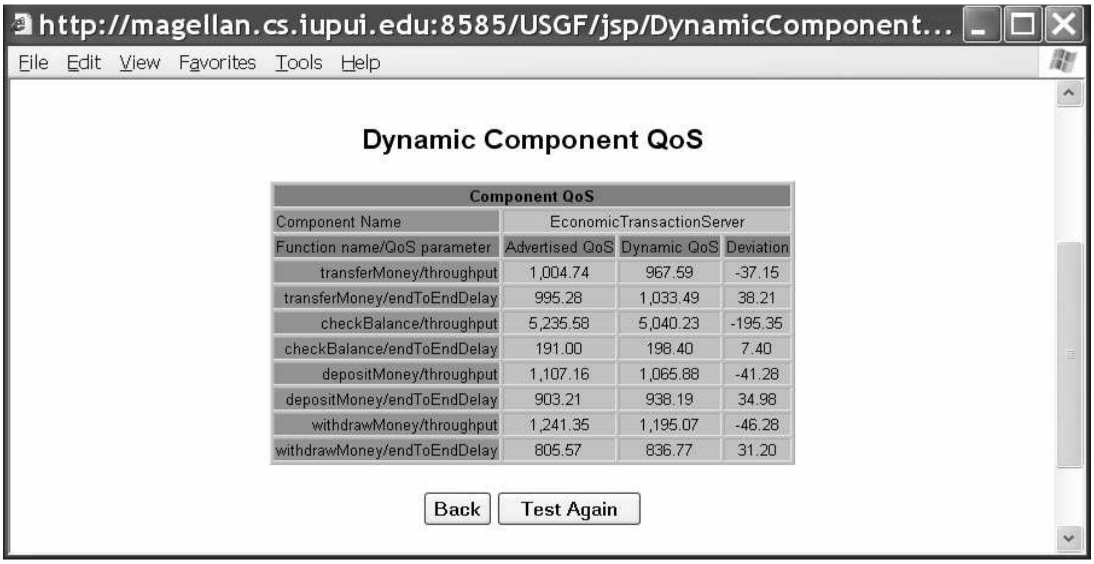


Figure 7.11 View Provided by *DynamicComponentQoS.jsp*

7.3.3.1.9 View Provided by *StaticSystemValidation.jsp*

Figure 7.12 illustrates the view provided by *StaticSystemValidation.jsp*. This view displays the results of the system QoS predicted by the QoS Composition Model. The QoS values under the column of *Expected System QoS* are the user QoS requirements. The predicted values are indicated under the column called *Predicted System QoS*. The difference between the expected and predicted system QoS is shown under the column called *Deviation*.

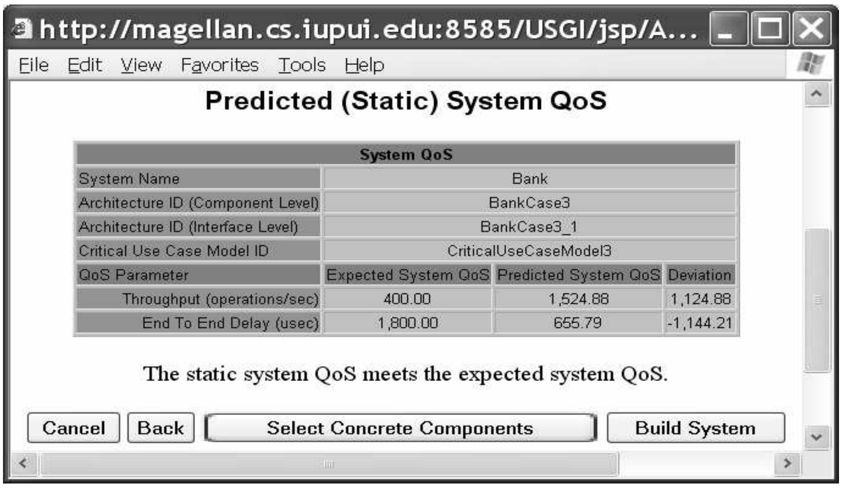


Figure 7.12 View Provided by *StaticSystemValidation.jsp*

7.3.3.1.10 View Provided by *DynamicSystemValidation.jsp*

Figure 7.13 illustrates the view provided by *DynamicSystemValidation.jsp*. This view displays the results of dynamic (or real) system QoS computed by executing the system, collecting the event traces and analyzing them. In the current prototype, the principles of the event grammars and event traces are not implemented. The dynamic QoS obtained using the current prototype contains simple pre-coded instrumentations that empirically measure the values of the QoS parameters for the integrated system. The QoS values under the column of *Expected System QoS* are the user QoS requirements. The QoS values obtained by empirical testing are shown under the column called *Dynamic*

System QoS. The difference between the expected and dynamic system QoS is shown under the column called *Deviation*.

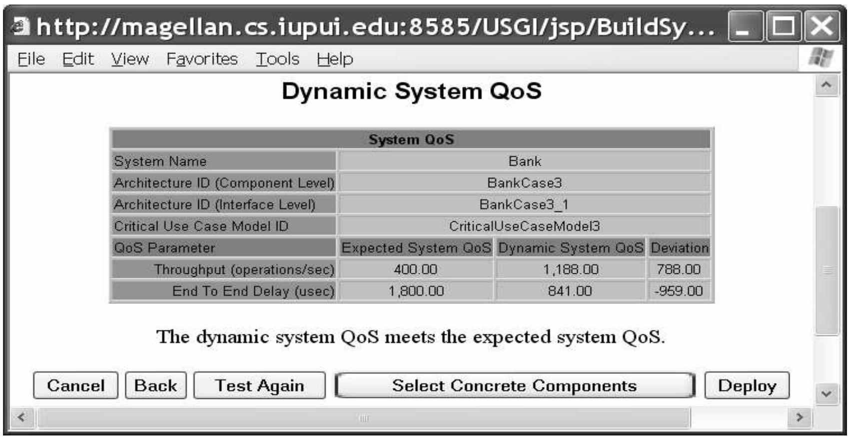


Figure 7.13 View Provided by *DynamicSystemValidation.jsp*

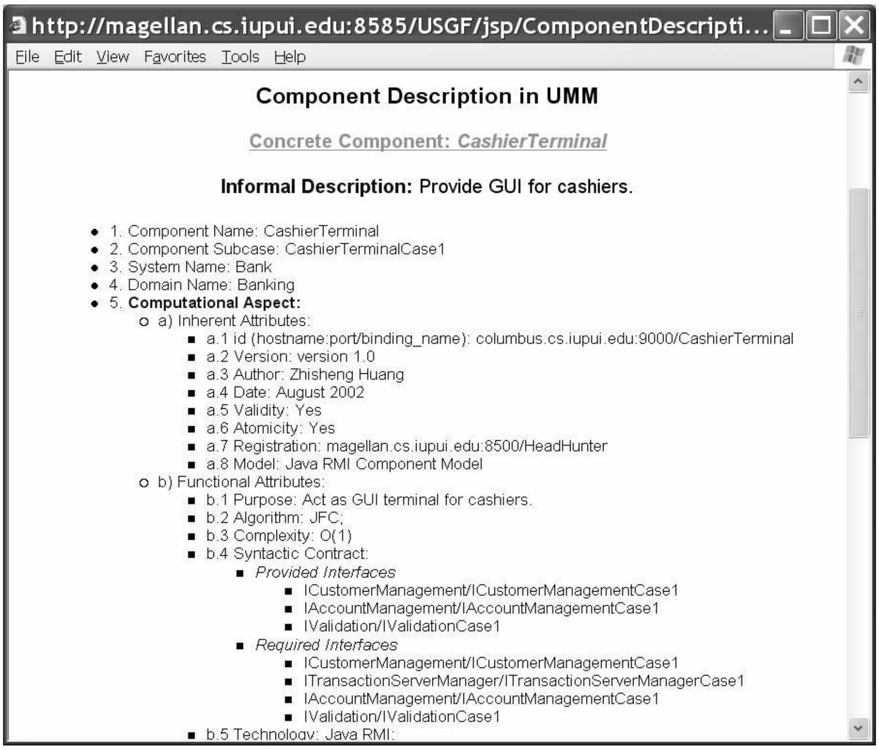


Figure 7.14 View Provided by *CompoenntDescription.jsp*

7.3.3.1.11 View Provided by *ComponentDescirption.jsp*

Figures 7.14 and 7.15 illustrate the view provided by *ComponentDescription.jsp*. This view displays the UMM specification for a component. The component can either be an abstract component or a concrete component. The difference between the UMM specification of an abstract component and that of a concrete component is discussed in Chapter 3.

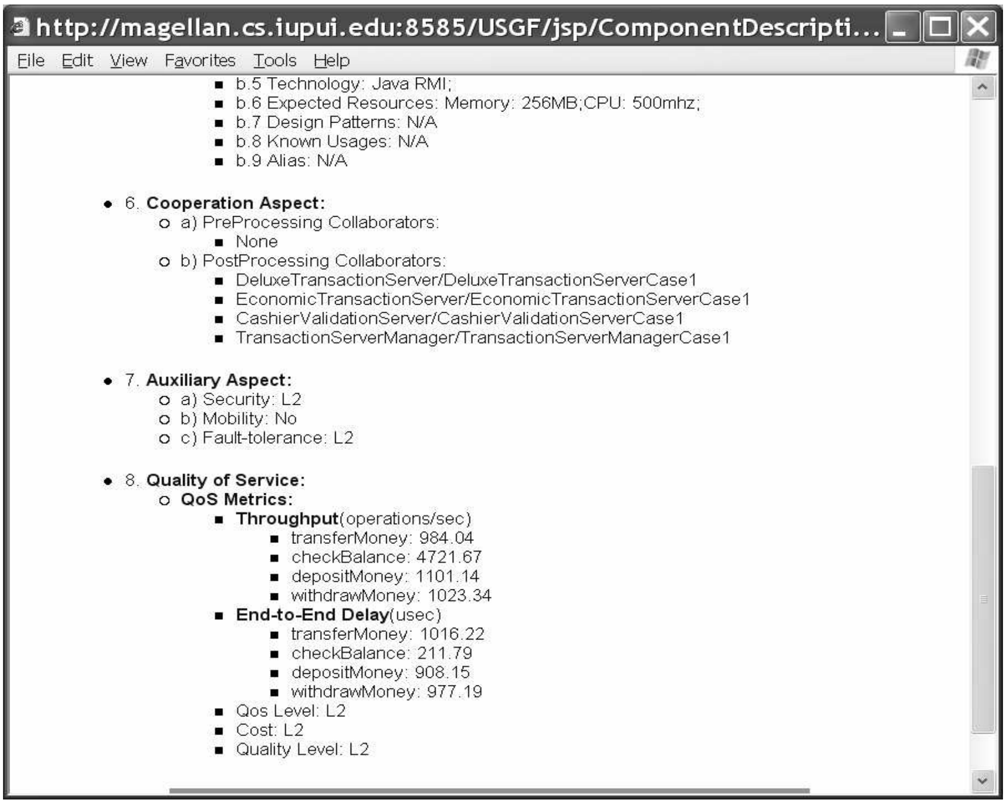


Figure 7.15 View Provided by *ComponentDescription.jsp* (Continued from Figure 7.14)

7.3.3.1.12 View Provided by *UGDMKBGeneration.jsp*

Figure 7.16 illustrates the view provided by *UGDMKBGeneration.jsp*. The view displays a set of choices of different XML parsers to translate UGDM models from XML format into Oracle database. In the current prototype, seven parsers are available:

Abstract Component Model Parser (for UMM Specification), *AMDNF (Component Level) Parser*, *AMDNF (Function/Interface Level) Parser*, *AMDNF Mapping Parser (Component Level to Function/Interface Level)*, *Component Interaction Parser*, *AMDNF and CUCM Mapping Parser (Function/Interface Level)* and *Abstract Component Interface Model Parser*. Details about each model are in Chapter 4 and Chapter 5. The option of *Reset Banking UGDMKB* refreshes the UGDMKB with the UGDM of the banking domain in XML formats by the above parsers.

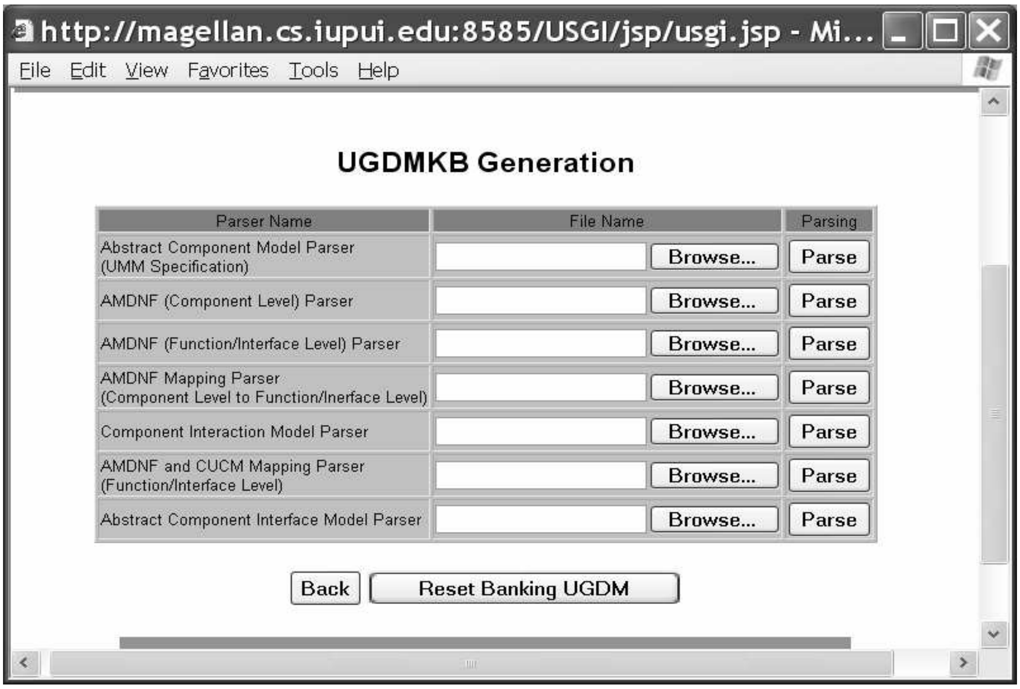
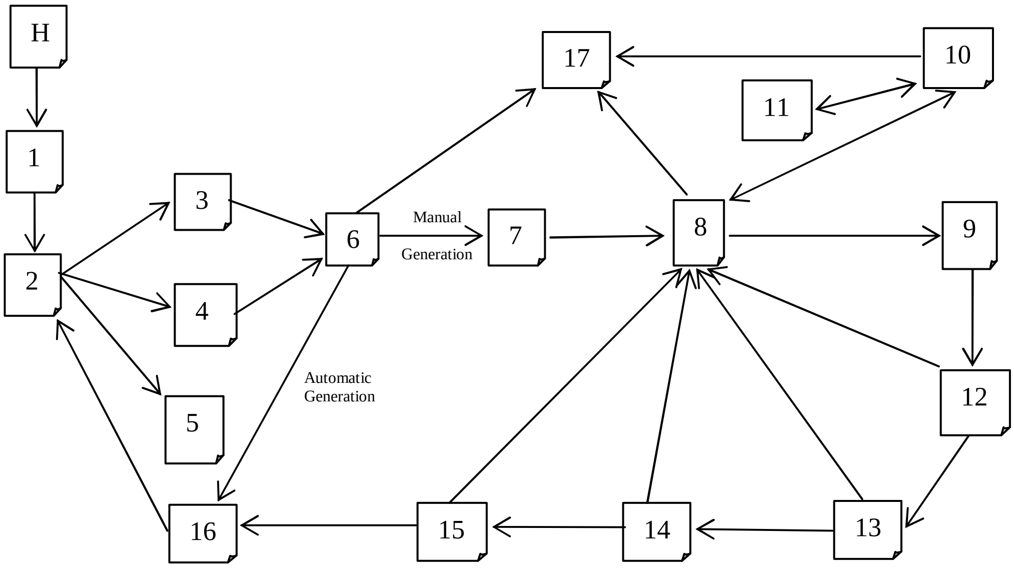


Figure 7.16 View Provided by *UGDMKBGeneration.jsp*

<u>7.3.3.2 Web Tier</u>

The *Web Tier* consists of *Web components*, which are the controllers in the MVC architecture. The controllers are responsible for coordinating the model and the view. The *Web Tier* in the USGI also consists of several proxy classes, which help the connection between *Web Components* in this tier and *Business Components* in the *Business Tier*.



Legends	
H: home.html	9: DetermineAdapterTypes.jsp
1: initiation.jsp	10 SelectConcreteComponents.jsp
2: usgi.jsp	11. DynamicComponentQoS.jsp
3: OrderWithoutNLP.jsp	12: AcquireAdapters.jsp
4: OrderWithNLP.jsp	13: StaticSystemValidation.jsp
5: UGDMKBGeneration.jsp	14: BuildSystem.jsp
6: Order.jsp	15: DynamicSystemValidation.jsp
7: AcquireConcreteComponents.jsp	16: Deploy.jsp
8: AvailableConcreteComponents.jsp	17: ComponentDescription.jsp

Figure 7.17 Flow between jsp Files in USGI Implementation

7.3.3.2.1 Web Components

The *Web Components* in the USGI prototype implementation are JSP pages. Figure 7.17 shows the major JSP pages in the prototype and their interactions to present the views in a logical way to users. In the view provided by *Order.jsp*, the users have the choices of the automatic system generation or the manual system generation. These choices make the biggest difference in the flow of control in the JSP pages.

- *home.html*: This is NOT a JSP page. This html page serves as the starting point of the USGI. It automatically redirects the control to *initiation.jsp*.

- *initiation.jsp:* This JSP page does not provide a view. The purpose of this JSP page is to initialize a session for a user. Objects of *OrderProcessorProxy*, *SystemGeneratorProxy* and *UGDMKBGeneratorProxy* are created and maintained in the session bean to be used throughout a session. These proxies are the connections between the *Web Tier* and the *Business Tier*. They are discussed in the next section.
- *usgi.jsp*: This JSP page provides the top-level choices available in the prototype: *System Generation without NLP*, *System Generation with NLP* and *UGDMKB Generation.*
- *OrderWithoutNLP.jsp*: This JSP page provides the starting interface to order a system from the banking domain example without the natural language processing support. It gathers information about an order in an order bean and passes the bean to *Order.jsp*.
- *OrderWithNLP.jsp*: This JSP page provides the starting interface to place an order for a system in a natural-language-like format. It shows an example order and shows to the users whether an order is understood by the system or not. It passes the natural-language-like order to the *NLP* in the business tier via the *NLPProxy*. It passes the order bean return from the *NLP* to *Order.jsp*.
- *UGDMKBGeneration.jsp*: This JSP page provides an access to a set of XML parsers to parse information about a GDM in the XML format into an Oracle database. It accesses the *UGDMKBGenerator* in the business tier via the *UGDMKBGeneratorProxy*.
- *Order.jsp*: This JSP page accepts order beans from either *OrderWithoutNLP.jsp* or *OrderWithNLP.jsp* and passes the order bean to the *OrderProcessor* in the business tier via the *OrderProcessorProxy*. It displays information about a system specification returned from the *OrderProcessor* and passes the system specification to *AcquireConcreComponents.jsp*.
- *AcquireConcreteComponents.jsp*: This JSP page accepts the system specification from *Order.jsp* and contacts the *SystemGenerator* in the *Buiness Tier* via the *SystemGeneratorProxy* to obtain a list of concrete components for the required abstract components indicated in the system specification. Then it passes the list of concrete components to *AvailableConcreteComponents.jsp*.

- *AvailableConcreteComponents.jsp*: This JSP page provides the view of the lists of concrete components that meets the requirements of the corresponding abstract component. When the users click on the “Found” link, the page is forwarded to *SelectConcreteCompoennts.jsp*.
- *SelectConcreteComponents.jsp*: This JSP page displays all the concrete components for an abstract component and prompts the users to select the required number of concrete components from the available concrete components.
- *DynamicComponentQoS.jsp*: This JSP page allows the dynamic testing of the QoS for a concrete component to check if the advertised QoS is accurate or not. It compares the results of the dynamic testing with the advertised values. If the dynamic testing results are different from the advertised values, no corrective actions are possible in the current prototype.
- *DetermineAdapterTypes.jsp*: When a set of concrete components are selected for generating a DCS, this JSP page functions as an entry point to determine what kind of adapters are needed if the selected concrete components are heterogeneous. It does so by contacting the *SystemGenerator* in the *Business Tier* via the *SystemGeneratorProxy* in this tier.
- *AcquireAdapters.jsp*: If any adapter is needed to bridge the heterogeneous concrete components, this JSP page contacts the *SystemGenerator* in the *Business Tier* via the *SystemGeneratorProxy* to get the necessary adapters.
- *StaticSystemValidation.jsp*: This JSP is the starting point for static system QoS validation, which is done by using the QoS Composition Model through the *SystemGenerator* in the *Business Tier* via the *SystemGeneratorProxy*. It displays the static validation results.
- *BuildSystem.jsp*: This JSP page is the starting point for configuring a system through the *SystemGenerator* in the *Business Tier* via the *SystemGeneratorProxy*. It displays to the users whether the configuration is successful or not.
- *DynamicSystemValidation.jsp*: This JSP page is the starting point for the dynamic validation of the system QoS and displays the validation results. It invokes the *SystemGenerator* in the *Business Tier* via the *SystemGeneratorProxy*.

- *Deploy.jsp*: This JSP page displays the complete information about an integrated system, including its order criteria, system specification, and static and dynamic QoS validation value, etc.
- *ComponentDescription.jsp*: This JSP page provides a detailed view that describes the UMM specification about an abstract component or a concrete component. The UMM specification is described in Chapter 3.

7.3.3.2.2 Proxy Classes

The proxy classes serve as the connectors between the *Web Tier* and the *Business Tier* in the USGI. There are four proxy classes in the USGI prototype implementation.

- *UGDMKBGeneratorProxy*: This proxy class connects *UGDMKBGenerator.jsp* in the *Web Tier* with the *UGDMKBGenerator* in the *Business Tier*.
- *OrderProcessorProxy*: This proxy class connects *Order.jsp* in the *Web Tier* with the *OrderProcessor* in the *Business Tier*.
- *SystemGeneratorProxy*: This proxy class connects many JSP pages in the *Web Tier* with the SystemGenerator in the *Business Tier*. The JSP pages connected by this proxy class include: *AcquireConcreteComponents.jsp*, *DynamicComponentQoS.jsp*, *DetermineAdapterTypes.jsp*, *AcquiredAdapters.jsp*, *StaticSystemValidation.jsp*, *BuildSystem.jsp*, *DynamicSystemValidation.jsp*.
- *NLPProxy*: This proxy class connects *OrderWithNLP.jsp* in the *Web Tier* with the *NLP* in the *Business Tier*.

7.3.3.2.3 Managing the State of a Session

Every user needs to track the information associated with the user requests and the associated responses. In the JSP pages, an http session object maintains JavaBeans that are specific to a user. The following state information is maintained.

- *Order Requirements*: The order requirements for a system are captured in an *OrderBean* and passed on to the *OrderProcessor* in the *Business Tier* via the *OrderProcessorProxy*.
- *System Specifications*: The details of a system that can satisfy the order requirements placed by a user include the system architecture ID at both component level and function/interface level, the system critical use case model ID, expected system QoS, expected component QoS and required abstract components. This information is captured in a JavaBean named *SystemSpecification*.
- *System Blueprints*: The complete information about an integrated system consists of the order information and the system specification for the system as well as the static and dynamic system validation results for the system, and the deployment information. All this information is captured in a JavaBean named *SystemBlueprint*.

7.3.3.3 Business Tier

The *Business Tier* consists of *Business Components*, which are a part of the *Model* in the MVC architecture in the USGI prototype design. *Business Components* here refers to standalone software units that provide services to components in other tiers or in the same tier. The service provided could be a computational effort or an access to underlying resources. *Business Components* can be remotely accessed using standard communication protocols. The *Business Components* in the USGI prototype include *UGDMKBGenerator*, *OrderProcessor*, *NLP*, *SystemGenerator*, *URDS_Proxy*, and *WrapperGlueGenerator_Proxy*.

7.3.3.3.1 UGDMKBGenerator

Figure 7.18 shows the class diagram of the *UGDMKBGenerator* and the interface it implements. The *UGDMKBGenerator* is invoked by the *UGDMKBGeneratorProxy* in the *Web Tier*. The details of the *UGDMKBGenerator* and the associated interface are provided below.

IUGDMKBGenerator: This remote interface publishes two methods.

- *parse()*: The purpose of this method is to translate a file in the XML format, which contains a model in the UGDM for a DCS domain, into an Oracle database.
- *resetBankingUGDM()*: The purpose of this method is to reset the banking domain example in the Oracle database in case the information in the database is corrupted.

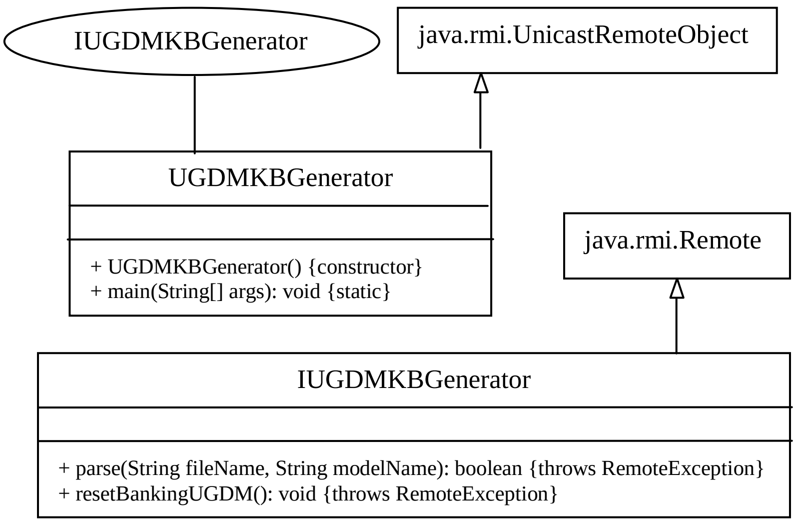


Figure 7.18 Class Diagram for *UGDMKBGenerator*

UGDMKBGenerator: This class implements the remote interface *IUGDMKBGenerator*. Currently the *UGDMKBGenerator* only implements part of the tasks outlined in Section 6.3.6. It only translates the models from the XML format, which is defined for each model in Appendix I, into the database by a set of XML parsers. These XML parsers use Apache's DOM parser technology [APA03]. The XML parsers that are used by the *UGDMKBGenerator* include:

- *UMMSepcification_XMLParser*: This parser translates a UMM specification of an abstract component from the XML format into an Oracle database.

- *Architecture_Component_XMLParser*: This parser translates an architecture model in disjunctive normal form at component level from the XML format into an Oracle database.
- *Architecture_Interface_XMLParser*: This parser translates an architecture model in disjunctive normal form at function/interface level from the XML format into an Oracle database.
- *Map_Architectures_XMLParser*: This parser translates an architecture model mapping from the XML format into an Oracle database.
- *Component_Interaction_XMLParser*: This parser translates a component interaction model from the XML format into an Oracle database.
- *Map_Architecture_CUCM_XMLParser*: This parser translates the mapping between the architecture model in disjunctive normal form and the critical use case model from the XML format into an Oracle database.
- *AbstractComponentInterface_XMLParser*: This parser translates an abstract component interface model from the XML format into an Oracle database.

7.3.3.3.2 Order Processor

Figure 7.19 shows the class diagram of *OrderProcessor* and the interface it implements. The *OrderProcessor* is invoked by the *OrderProcessorProxy* in the *Web Tier*.

IOrderProcessor: This remote interface publishes one method.

- *order()*: The purpose of this method is to determine the system specification according to the user requirements by querying the database that stores the UGDM. Currently, there are two options for this method in the prototype implementation for the banking domain example: order with the natural language processing support and order without the natural language processing support.
- *Order_simulation()*: This method has the same functionality as the one above. The difference is that this method queries through local data

structures that simulate the functionality of the database that stores the UGDM.

OrderProcessor: This class implements the interface *IOrderProcessor*. The *OrderProcessor* implements the algorithm outlined in Section 3.7.8. The *OrderProcessor* uses the UGDM in the Oracle database. It also uses the library of the QoS composition and decomposition rules to derive the expected component QoS from the System QoS. The QoS library for the banking domain example is implemented in the class *QCDM_Bank.*

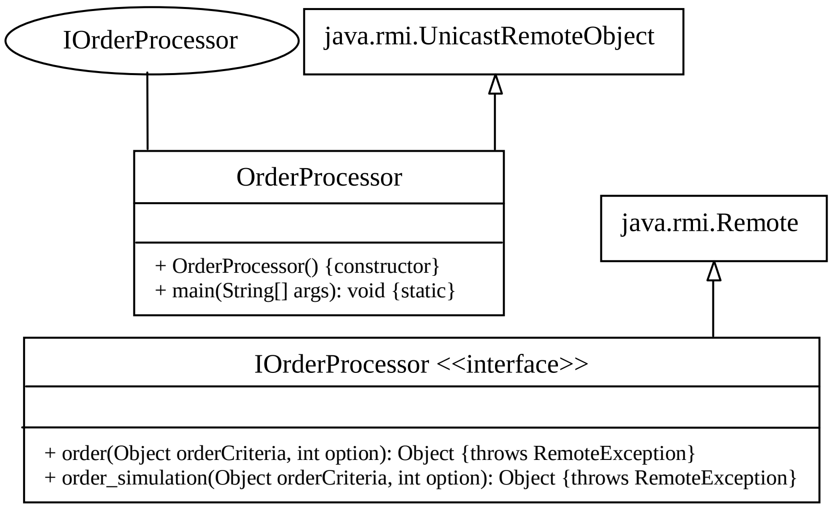


Figure 7.19 Class Diagram for *OrderProcessor*

7.3.3.3.3 System Generator

Figure 7.20 shows the class diagram of the *SystemGenerator* and the interface it implements. The *SystemGenerator* is invoked by the *SystemGeneratorProxy* in the *Web Tier*.

ISystemGenerator: This remote interface publishes 15 methods.

- *acquireConcreteComponents()*: This method takes a system specification as its input. The system specification contains a list of required abstract components. This method searches for each required abstract component through *QueryManager* of the URDS via the *URDS_Proxy*. It returns a *Hashtable* containing one list of concrete components found for each required abstract component. The keys for the *Hashtable* are the names of the abstract components.

- *acquireConcreteComponents_simulation()*: This method has the same functionality as the one above. However, instead of looking for the concrete components through the *URDS*, it looks for the concrete components in a local repository, which simulates the functionality of the *URDS*.

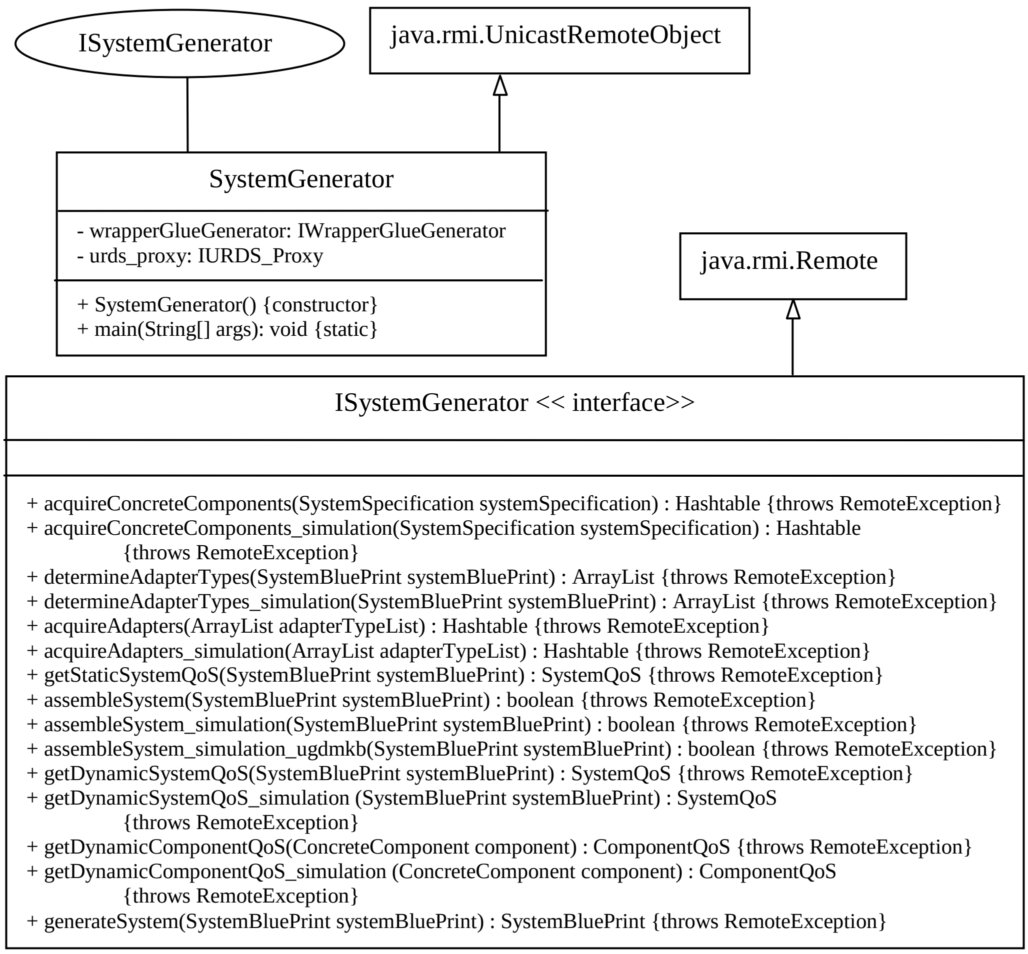


Figure 7.20 Class Diagram for *SystemGenerator*

- *determineAdapterTypes():* This method takes as its input a *SystemBluePrint* which consists of the selected concrete components for assembling a system. This method consults the UGDM in the database to determine what kinds of adapters are needed for assembling the system.
- *determineAdapterTypes_simulation()*: This method has the same functionality as the one above. However, instead of consulting the UGDM in the database, it consults local data structures that simulate the functionality of the database that stores the UGDM.
- *acquireAdapters()*: This method takes as its input a list of required adapters and acquires them through the URDS. If the URDS can not find an adapter, it sends the query for that adapter to the *WrapperGlueGenerator* via the *WrapperGlueGenerator_Proxy*.
- *acquireAdapters_simulation()*: This method has the same functionality as the one above. However, instead of looking for the adapters through the *URDS*, it looks for the adapters in a local repository which simulates the functionality of the *URDS*.
- *getStaticSystemQoS()*: This method takes as its input a *SystemBlueprint* which consists of all the information necessary for assembling a system. It calculates static system QoS by using QoS composition rules in the *QCDM_Bank* library.
- *assembleSystem()*: This method takes as its input a *SystemBlueprint* which consists of all the information necessary for assembling a system and assembles the system by consulting the UGDM in the database.
- *assembleSystem_simulation()*: This method has the same functionality as the one above. However, it does nothing but simply returns information to indicate that the system was assembled successfully. This allows the demonstration of the prototype without running the banking components.
- *assembleSysem_simulation_ugdmkb()*: This method has the same functionality as the one above. However, instead of consulting the UGDM

in the database, it consults local data structures that simulate the functionality of the database that stores the UGDM.

- *getDynamicSystemQoS()*: This method takes as its input a *SystemBlueprint*. It dynamically calculates the system QoS using the methods such as the event traces. In this prototype, the event traces are not implemented. The dynamic system QoS is measured by the pre-coded instructions.
- *getDynamicSystemQoS_simulation()*: This method has the same purpose as the one above. The difference is that this method only simulates the activity of dynamically getting the system QoS. The simulation is done through the random number generation.
- *getDynamicComponentQoS()*: This method takes as its input a *ConcreteComponent*. It dynamically calculates the component QoS using methods such as event traces. In this prototype, the event traces are not implemented. The dynamic component QoS is measured by the pre-coded instructions.
- *getDynamicComponentQoS_simulation()*: This method has the same purpose as the one above. The difference is that this method only simulates the activity of dynamically getting the component QoS. The simulation is done through the random number generation.
- *generateSystem()*: The purpose of this method is to generate a system automatically. It implements the system generation process outlined in Chapter 6. It achieves its purpose by a sequence of calls to other methods defined in this interface.

SystemGenerator: This class implements the *ISystemGenerator*. The *SystemGenerator* implements the algorithms outlined in Section 6.3.9. The *SystemGenerator* uses the UGDM in the Oracle database. It also uses the library of the QoS composition and decomposition rules to predict the static system QoS. The QoS library for the banking domain example is implemented in the class *QCDM_Bank*.

7.3.3.3.4 URDS_Proxy

Figure 7.21 shows the class diagram of the *URDS_Proxy* and the interface it implements. The *URDS_Proxy* is invoked by the *SystemGenerator* in the *Business Tier*. The *URDS_Proxy* accesses the URDS already implemented in the UniFrame research.

IURDS_Proxy: This remote interface publishes two methods.

- *searchConcreteComponents()*: This method takes as its input an abstract component. It prepares a *QueryBean* for the abstract component and sends the query to the *QueryManager* of the URDS. It returns a list of concrete components found for the abstract component.
- *searchConcreteComponents_simulation()*: This method has the same function as the one above. However, instead of looking for the concrete components through the URDS, it looks for concrete components in a local repository which simulates the functionality of the URDS.

URDS_Proxy: This class implements the *IURDS_Proxy*. It accesses the URDS through the interface published by the *QueryManager* in the URDS.

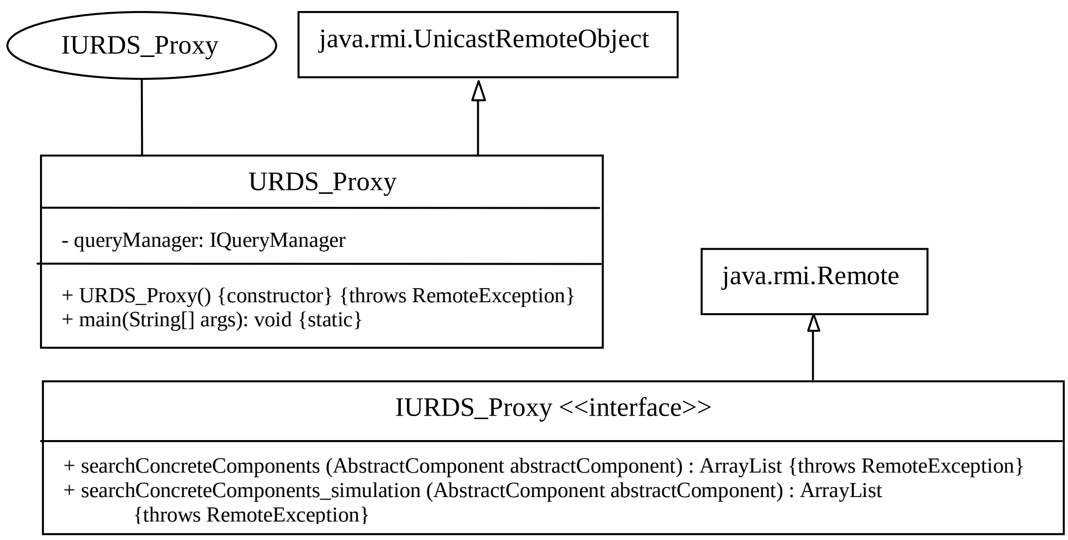


Figure 7.21 Class Diagram for *URDS_Proxy*

7.3.3.3.5 Natural Language Processor

Figure 7.22 shows the class diagram for the Natural Language Processor (NLP) and the interface it implements. The NLP is invoked by *the NLP_Proxy* in the *Web Tier*.

INLP: This remote interface publishes one method.

- *order()*: This method sends the natural-language-like system requirements as specified in the argument to the natural language processing service. It returns the system specification as an *OrderBean*.

NLP: This class implements the *INLP*. The *NLP* itself is a proxy that accesses the natural language processing service created by the collaborators of the UniFrame research at University of Alabama at Birmingham [LEE02a]. The natural language processing service implemented for this prototype is for the banking domain example.

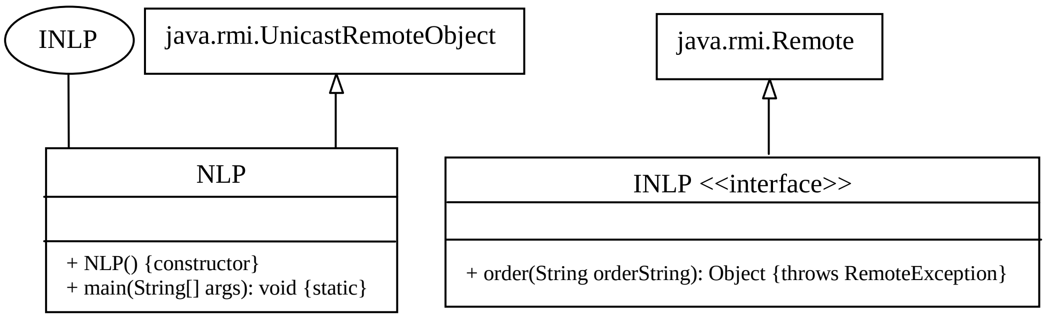


Figure 7.22 Class Diagram for *NLP*

7.3.3.3.6 WrapperGlueGenerator_Proxy

Figure 7.23 shows the class diagram for the *WrapperGlueGenerator_Proxy* and the interface it implements. The *WrapperGlueGenerator_Proxy* is invoked by the *SystemGenerator* in the *Business Tier*. The *WrapperGlueGenerator_Proxy* accesses the *WrapperGlueGenerator* being implemented by University of Alabama at Birmingham [CAO02, ZHA02], a collaborator of the UniFrame research.

IWrapperGlueGenerator_Proxy: This remote interface publishes one method.

- *generateWrapperGlue()*: This method takes as its argument an *AdapterType* and forwards the request to the *WrapperGlueGenerator* to generate the required adapter.

WrapperGlueGenerator_Proxy: *IWrapperGlueGenerator_Proxy* is implemented by this class. It is a proxy to access the Wrapper and Glue Generator service being implemented by our collaborator at University of Alabama at Birmingham [CAO02, ZHA02].

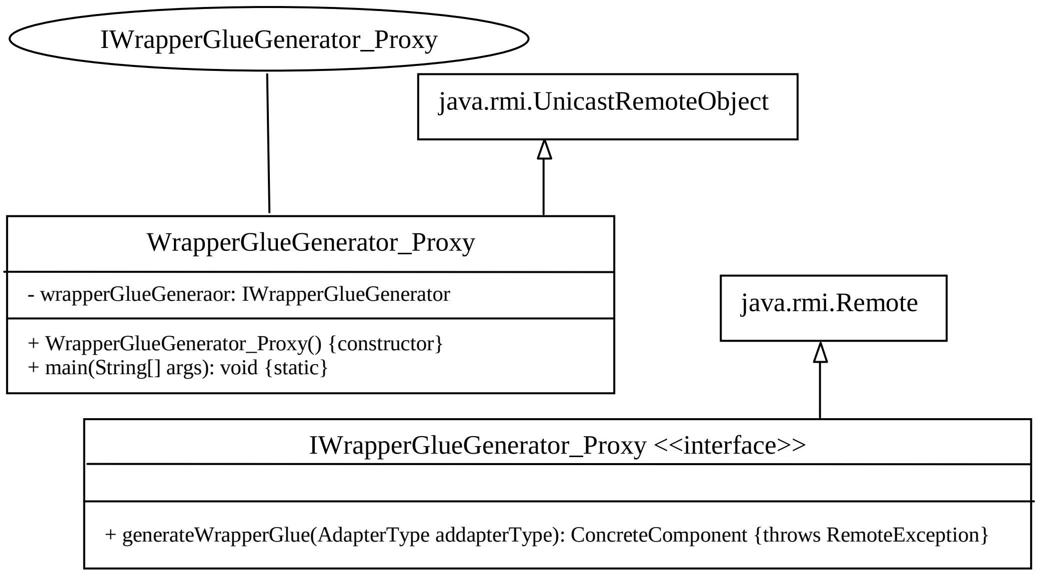


Figure 7.23 Class Diagram for *WrapperGlueGenerator_Proxy*

<u>7.3.3.4 Database Tier</u>

The *Database Tier* is responsible for storing the persistent data in the USGI. The persistent data in the USGI is the UGDM. The USGI maintains persistent data in an Oracle database (version 8.1.7) which is a relational database. The data are stored in database tables. These database tables store information about various models of the UGDM. The creation and maintenance of the database tables are done by the

UGDMKBGenerator. The UGDM information in the database is used by the *OrderProcessor* and the *SystemGenerator* in the USGI. The database is accessed and updated through the JDBC technology.

7.3.3.4.1 Schemas for the Abstract Component Model

Figure 7.24 illustrates the schemas for the abstract component model in the UGDM. The schemas consist of twelve tables: *UMMSpecification*, *Algorithms*, *RequiredInterfaces*, *ProvidedInterfaces*, *Technologies*, *ExpectedResources*, *DesignPatterns*, *KnownUsages*, *Aliases*, *PreprocessingCollaborators*, *PostprocessingCollaborators* and *QoSMetrics*. Information about each abstract component is stored in these twelve tables. The information reflects the UMM specification (details are in Chapter 3) of a component in the UniFrame Approach.

7.3.3.4.1.1 UMMSpecification Table

The *UMMSpecification* table holds entries from a UMM specification that has no more than one value for each abstract component. The columns in this table include those that can identify an abstract component, such as *ComponentName*, *DomainName* and *SystemName*, and those that are attributes of an abstract component, such as, *Description*, *HostID*, *Version*, *Author*, *CreationDate*, *Validity*, *Atomicity*, *Registration*, *Model*, *Purpose*, *Complexity*, *Mobility*, *Security*, *FaultTolerance*, *QoSLevel*, *Cost* and *QualityLevel*. Each abstract component has exactly one entry in this table. An example of a record of this table is <’AccountDatabase’, ‘Banking’, ‘Bank’, ‘Provide an account database service.’, ‘N/A’, ‘version 1.0’, ‘N/A’, ‘N/A’, ‘N/A’, ‘Yes’, ‘N/A, ‘N/A’, ‘Serve as an account database.’, ‘N/A’, ‘No’, ‘L0’, ‘L0’, ‘N/A’, ‘N/A’, ‘N/A’>. The first three entries in this example are the component name, domain name and system name respectively. The rest are the values for the attributes stated above respectively.

Schema for UMMSecification

Column Name	Column Type
ComponentName	VARCHAR
DomainName	VARCHAR
SystemName	VARCHAR
Description	VARCHAR
HostID	VARCHAR
Version	VARCHAR
Author	VARCHAR
CreatingDate	VARCHAR
Validity	VARCHAR
Atomicity	VARCHAR
Registration	VARCHAR
Model	VARCHAR
Purpose	VARCHAR
Complexity	VARCHAR
Mobility	VARCHAR
Security	VARCHAR
FaultTolerance	VARCHAR
QoSLevel	VARCHAR
Cost	VARCHAR
QualityLevel	VARCHAR

Schema for Algorithms

Column Name	Column Type
ComponentName	VARCHAR
DomainName	VARCHAR
SystemName	VARCHAR
Algorithm	VARCHAR

Schema for RequiredInterfaces

Column Name	Column Type
ComponentName	VARCHAR
DomainName	VARCHAR
SystemName	VARCHAR
Interface	VARCHAR

Schema for ProvidedInterfaces

Column Name	Column Type
ComponentName	VARCHAR
DomainName	VARCHAR
SystemName	VARCHAR
Interface	VARCHAR

Schema for Technologies

Column Name	Column Type
ComponentName	VARCHAR
DomainName	VARCHAR
SystemName	VARCHAR
Technology	VARCHAR

Schema for ExpectedResources

Column Name	Column Type
ComponentName	VARCHAR
DomainName	VARCHAR
SystemName	VARCHAR
ExpectedResource	VARCHAR

Schema for DesignPatterns

Column Name	Column Type
ComponentName	VARCHAR
DomainName	VARCHAR
SystemName	VARCHAR
Pattern	VARCHAR

Schema for KnownUsages

Column Name	Column Type
ComponentName	VARCHAR
DomainName	VARCHAR
SystemName	VARCHAR
Usage	VARCHAR

Schema for Aliases

Column Name	Column Type
ComponentName	VARCHAR
DomainName	VARCHAR
SystemName	VARCHAR
Alias	VARCHAR

Schema for PreprocessingCollaborators

Column Name	Column Type
ComponentName	VARCHAR
DomainName	VARCHAR
SystemName	VARCHAR
Collaborator	VARCHAR

Schema for PostprocessingCollaborators

Column Name	Column Type
ComponentName	VARCHAR
DomainName	VARCHAR
SystemName	VARCHAR
Collaborator	VARCHAR

Schema for QoSMetrics

Column Name	Column Type
ComponentName	VARCHAR
DomainName	VARCHAR
SystemName	VARCHAR
Metric	VARCHAR

Figure 7.24 Schemas for Abstract Component Model in the UGDM

7.3.3.4.1.2 Algorithms Table

The *Algorithms* table holds the possible algorithms that may be used to implement the concrete components for an abstract component. The columns in this table include those that can identify an abstract component, such as *ComponentName*, *DomainName* and *SysemName*, and another one for the name of a possible algorithm. One abstract component can have multiple entries in this table. An example of a record for this table is <'AccountDatabase', 'Banking', 'Bank', 'Merge Sort'>. The first three entries in this example identify the abstract component and the last entry indicates that the Merge Sort algorithm can be used to implement the concrete components for this abstract component.

7.3.3.4.1.3 RequiredInterfaces Table

The *RequiredInterfaces* table holds the required interfaces for an abstract component. The columns in this table include those that can identify an abstract component, such as *ComponentName*, *DomainName* and *SysemName*, and another one for the name of a required interface. One abstract component can have multiple entries in this table. An example of a record for this table is <'DeluxeTransactionServer', 'Banking', 'Bank', 'IAccountDatabaseCase1'>. The first three entries in this example identify the abstract component and the last entry indicates that the interface *IAcountDatabaseCase1* is required by this abstract component.

7.3.3.4.1.4 ProvidedInterfaces Table

The *ProvidedInterfaces* table holds the provided interfaces for an abstract component. The columns in this table include those that can identify an abstract component, such as *ComponentName*, *DomainName* and *SysemName*, and another one for the name of a provided interface. One abstract component can have multiple entries in this table. An example of a record for this table is <'AccountDatabase', 'Banking', 'Bank', 'IAccountDatabaseCase1'>. The first three entries in this example identify the abstract component and the last entry indicates that the interface *IAcountDatabaseCase1* is provided by this abstract component.

7.3.3.4.1.5 Techonologies Table

The *Technologies* table holds the possible technologies that may be used to implement the concrete components for an abstract component. The columns in this table include those that can identify an abstract component, such as *ComponentName*, *DomainName* and *SysemName*, and another one for the name of the possible technology that can be used to implement the abstract component. One abstract component can have multiple entries in this table. An example of a record for this table is <'AccountDatabase', 'Banking', 'Bank', 'Java RMI'>. The first three entries in this example identify the abstract component and the last entry indicates that Java RMI may be used to implement the concrete components for this abstract component.

7.3.3.4.1.6 ExpectedResources Table

The *ExpectedResources* table holds the possible resources that may be required by the concrete components of an abstract component. The columns in this table include those that can identify an abstract component, such as *ComponentName*, *DomainName* and *SysemName*, and another one for the name of the possible resource. One abstract component can have multiple entries in this table. An example of a record for this table is <'AccountDatabase', 'Banking', 'Bank', 'Memory'>. The first three entries in this example identify the abstract component and the last entry indicates that the memory is a possible required resource by the concrete components for this abstract component.

7.3.3.4.1.7 DesignPatterns Table

The *DesignPatterns* table holds the possible design patterns that may be applied to implement the concrete components for an abstract component. The columns in this table include those that can identify an abstract component, such as *ComponentName*, *DomainName* and *SysemName*, and another one for the name of the possible design pattern. One abstract component can have multiple entries in this table. An example of a record for this table is <'AccountDatabase', 'Banking', 'Bank', 'Factory Pattern'>. The first three entries in this example identify the abstract component and the last one indicates that the factory pattern may be used to implement the concrete components for the abstract component.

7.3.3.4.1.8 KnownUsages Table

The *KnownUsages* table holds the known application of an abstract component. The columns in this table include those that can identify an abstract component, such as *ComponentName*, *DomainName* and *SysemName*, and another one for the name of the area that the abstract component is used. One abstract component can have multiple entries in this table. An example of a record for this table is <'AccountDatabase', 'Banking', 'Bank', 'Finance'>. The first three entries in this example identify the abstract component and the last one indicates that this abstract component has been used in the area of finance.

7.3.3.4.1.9 Aliases Table

The *Aliases* table holds the possible aliases of an abstract component. The columns in this table include those that can identify an abstract component, such as *ComponentName*, *DomainName* and *SysemName*, and another one for the name of a possible alias for the abstract component. One abstract component can have multiple entries in this table. An example of a record for this table is <'AccountDatabase', 'Banking', 'Bank', 'AccountRepository'>. The first three entries in this example identify the abstract component and the last one indicates that *AccountRepository* is another name for this abstract component.

7.3.3.4.1.10 PreprocessingCollaborators Table

The *PreprocessingCollaborators* table holds the preprocessing collaborators of an abstract component. The columns in this table include those that can identify an abstract component, such as *ComponentName*, *DomainName* and *SysemName*, and another one for the name of a preprocessing collaborator. One abstract component can have multiple entries in this table. An example of a record for this table is <'AccountDatabase', 'Banking', 'Bank', 'DeluxeTransactionServer'>. The first three entries in this example identify the abstract component and the last one indicates that *DeluxeTransactionServer* is a preprocessing collaborator of this abstract component.

7.3.3.4.1.11 PostprocessingCollaborators Table

The *PostprocessingCollaborators* table holds the postprocessing collaborators of an abstract component. The columns in this table include those that can identify an abstract component, such as *ComponentName*, *DomainName* and *SysemName*, and another one for the name of a postprocessing collaborator. One abstract component can have multiple entries in this table. An example of a record for this table is <'DeluxeTransactionServer', 'Banking', 'Bank', 'AccountDatabase'>. The first three entries in this example identify the abstract component and the last one indicates that AccountDatabase is a postprocessing collaborator of this abstract component.

7.3.3.4.1.12 QoSMetrics Table

The *QoSMetrics* table holds the QoS metrics of an abstract component that must be validated when implemented. The columns in this table include those that can identify an abstract component, such as *ComponentName*, *DomainName* and *SysemName*, and another one for the name of a QoS metric. One abstract component can have multiple entries in this table. An example of a record for this table is <'AccountDatabase', 'Banking', 'Bank', 'throughput'>. The first three entries in this example identify the abstract component and the last one indicates that *throughput* is a QoS parameter that must be validated when the concrete components of the abstract component are implemented.

Schema for ArchitectureComponent

Column Name	Column Type
SystemName	VARCHAR
CaseName	VARCHAR
ComponentName	VARCHAR

Schema for ArchitectureInterface

Column Name	Column Type
SystemName	VARCHAR
CaseName	VARCHAR
ComponentName	VARCHAR
ComponentSubcase	VARCHAR

Schema for MapArchitectures

Column Name	Column Type
SystemName	VARCHAR
CaseNameFrom	VARCHAR
CaseNameTo	VARCHAR

Schema for ComponentInteraction

Column Name	Column Type
SystemName	VARCHAR
Initiator	VARCHAR
Responder	VARCHAR

Schema for AbstractComponentInterface

Column Name	Column Type
DomainName	VARCHAR
SystemName	VARCHAR
ComponentName	VARCHAR
ComponentSubcase	VARCHAR
InterfaceType	VARCHAR
InterfaceName	VARCHAR
InterfaceSubcase	VARCHAR

Schema for MapArchitectureCUCM

Column Name	Column Type
SystemName	VARCHAR
CaseNameFrom	VARCHAR
CaseNameTo	VARCHAR

Figure 7.25 Schemas for Other Models in the UGDM

7.3.3.4.2 Schema for the AMDNF at Component Level

The schema for the Architecture Model in Disjunctive Normal Form (AMDNF) at Component Level is shown in Figure 7.25 as the database table *ArchitectureComponent*. The columns in this table include *SystemName*, *CaseName* and *ComponentName*. *SystemName* and *CaseName* together identify a case in the AMDNF at component level in the UGDM. The *ComponentName* is the entry for a component that constitutes the case. Thus each case can have multiple entries in this table. An example of a record for this table is <’Bank’, ‘BankCase1’, ‘CashierTerminal’>. This example indicates that the component *CashierTerminal* is part of the case *BankCase1* of the *Bank* system.

7.3.3.4.3 Schema for the AMDNF at Function/Interface Level

The schema for the Architecture Model in Disjunctive Normal Form (AMDNF) at Function/Inerface Level is shown in Figure 7.25 as the database table *ArchitectureInterface*. The columns in this table include *SystemName*, *CaseName*, *ComponentName* and *ComponentSubcase*. *SystemName* and *CaseName* together identify a case in the AMDNF in the UGDM. The *ComponentName* is the entry for a component that constitutes the case. The *ComponentSubcase* reveals the special information about the interfaces of the component, such as the communication patterns. Each case can have multiple entries in this table. However, for each {SystemName, CaseName, ComponentName} triple, there is only one entry. An example of a record for this table is <’Bank’, ‘BankCase1_1’, ‘CashierTerminal’, ‘CashierTerminalCase1’>. This example indicates that the component *CashierTerminal* is part of the case *BankCase1_1* of the *Bank* system and the *CashierTerminalCase1* represents the interfaces of the *CashierTerminal*.

7.3.3.4.4 Schema for the Architecture Model Mapping

The schema for the Architecture Model Mapping is shown in Figure 7.25 as the database table *MapArchitectures*. The columns in this table include *SystemName*, *CaseNameFrom* and *CaseNameTo*. *SystemName* identifies a system. The *CaseNameFrom* indicates a case in the Architecture Model in Disjunctive Normal Form (AMDNF) at the component level. The *CaseNameTo* indicates a case in the AMDNF at the function/interface level. The mapping is from *CaseNameFrom* to *CaseNameTo*. Each {SystemName, CaseNameFrom} pair can have only one entry in the database table. The mapping is unidirectional. An example of a record for this table is <’Bank’, ‘BankCase1’, ‘BankCase1_1’>. This example indicates that for the *Bank* system, *BankCase1* (a case in the AMDNF at the component level) is mapped to *BankCase1_1* (a case in the AMDNF at the function/interface level).

7.3.3.4.5 Schema for Component Interaction Model

The schema for the Component Interaction Model is shown in Figure 7.25 as the database table *ComponentInteraction*. The columns in this table include *SystemName*, *Initiator* and *Responder*. *SystemName* identifies a system. The *Initiator* is the entry for the abstract component that initiates an interaction. The *Responder* is the entry for the abstract component that responds to the *Initiator*. For each *Initiator*, there can be multiple entries in this table. For each *Responder*, there can also be multiple entries in the table. If two components in an interaction are peer-to-peer, then there should be two entries in the database table for this kind of interaction. An example of a record for this table is <'Bank', 'DeluxeTransactionServer', 'AccountDatabase'>. This example indicates that in the *Bank* system, for the interaction between *DeluxeTransactionServer* and *AccountDatabase*, the former component is the initiator and the latter component is the responder.

7.3.3.4.6 Schema for Abstract Component Interface Model

The schema for the Abstract Component Interface Model is shown in Figure 7.25 as the database table *AbstractComponentInterface*. The columns in this table include *DomainName*, *SystemName*, *ComponentName*, *ComponentSubcase*, *InterfaceType*, *InterfaceName* and *InterfaceSubcase*. *DomainName*, *SystemName, ComponentName* and *ComponentSubcase* together identify an abstract component at the function/interface level. *InterfaceName* and *InterfaceSubcase* identify an interface. The *InterfaceType* indicates the type of the interface in the entries of *InterfaceName* and *InterfaceSubcase*. The value of *InterfaceType* is either *Required* or *Provided*. For each abstract component at function/interface level, there can be multiple entries in this table. An example of a record for this table is <'Banking', 'Bank', 'AccountDatabase', ‘AccountDatabaseCase1’, ‘Provided’, ‘IAccountDatabase’, ‘IAccountDatabaseCase1’>. The first four entries in this example identify the abstract component. The last two entries identify an interface and the fifth entry indicates that this interface is a provided interface of the component.

7.3.3.4.7 Schema for AMDNF and CUCM Mapping

The schema for the Architecture Model in Disjunctive Normal Form (AMDNF) and Critical Use Case Model (CUCM) Mapping at the function/interface level is shown in Figure 7.25 as the database table *MapArchitectureCUCM*. The columns in this table include *SystemName*, *CaseNameFrom* and *CaseNameTo*. *SystemName* identifies a system. The *CaseNameFrom* is the entry for a case in the AMDNF at function/interface level. *CaseNameTo* is the entry for a case in the Critical Use Case Model (CUCM) at function/interface level. Each {SystemName, CaseNameFrom} pair can have only one entry in the table. An example of a record for this table is <’Bank’, ‘BankCase1_1’, ‘CriticalUseCaseModel3’>. This example indicates that in the *Bank* system, *BankCase1_1* (a case in the AMDNF at function/interface level) is mapped to *CriticalUseCaseModel3* (a case in the CUCM at the function/interface level).

7.3.4 Experimental Results

This section provides the initial experimental results of using the USGPF to order simple bank DCS from the banking domain example developed in Chapter 5. There are two ordering schemes designed for this bank DCS family, one with the tabular ordering language and the other one with the natural-language-like ordering language.

7.3.4.1 Ordering Scheme with Tabular Ordering Language

The tabular ordering language designed for the banking domain example is shown in Table 5.39. The experiment was done with the following ordering criteria: 1) Bank

Type: Advanced Bank; 2) User Terminal (copy number): *ATM* (1 copy), *Cashier Terminal* (1 copy); 4) System QoS: throughput > 700 operations/second, end to end delay < 1500 microseconds. After placing the order, the *OrderProcessor* returned the following system specification:

```
System Name: Bank
Architecture ID (Component Level): BankCase1
Architecture ID (Interface Level): BankCase1_1
Critical Use Case Model ID: CriticalUseCaseModel3
Expected System QoS:
    Throughput (operations/second): 700.0
    End to end delay (microsecond): 1500.0
Expected Component QoS:
    EconomicTransactionServer:
        transferMoney/throughput (operations/second): > 700.0 microsecond
        transferMoney/endToEndDelay (microsecond): < 1500.0 operations/second
        depositMoney/throughput (operations/second): > 700.0 microsecond
        depositMoney/endToEndDelay (microsecond): <1500 operations/second
        withdrawMoney/throughput (operations/second): > 700.0 microsecond
        withdrawMoney/endToEndDelay (microsecond): < 1500.0 operations/second
    …
Required Abstract Components: (Component Name/Component Subcase/Copy Number)
    ATM/ATMCase1/1
    CashierTerminal/CashierTerminalCase1/1
    CashierValidationServer/CashierValidationServerCase1/1
    CustomerValidationSever/CustomerValidationServerCase1/1
    TransactionServerManager/TransactionServerMangerCase1/1
    EconomicTransactionServer/EconomicTransactionServerCase1/1
```

The above system specification defines a system architecture that meets the ordering requirements including the required abstract components, the expected system QoS and the expected component QoS derived from the expected system QoS by the QoS decomposition rules.

The next step is to generate the system based on this system specification. The experiment was done with the option, *Generate System (Manual)*, in order to interact with the system generation process. The system specification was sent to the *SystemGenerator*. The *SystemGenerator* firstly found the following concrete components for the required abstract components via the URDS (the component ID for each concrete component is listed):

ATM/ATMCase1:
1) magellan.cs.iupui.edu:9000/ATM
2) raleigh.cs.iupui.edu:9000/ATM
3) columnbus.cs.iupui.edu:9000/ATM

CashierTerminal/CashierTerminalCase1:
1) magellan.cs.iupui.edu:9000/ CashierTerminal
2) raleigh.cs.iupui.edu:9000/ CashierTerminal
3) columnbus.cs.iupui.edu:9000/ CashierTerminal

CashierValidationServer/CashierValidationServerCase1:
1) magellan.cs.iupui.edu:9000/ CashierValidationServer
2) raleigh.cs.iupui.edu:9000/ CashierValidationServer
3) columnbus.cs.iupui.edu:9000/ CashierValidationServer
4) *http://134.68.140.142:9000/CashierValidationServer*

CustomerValidationServer/CustomerValidationServerCase1
1) magellan.cs.iupui.edu:9000/ CustomerValidationServer
2) raleigh.cs.iupui.edu:9000/ CustomerValidationServer
3) columnbus.cs.iupui.edu:9000/ CustomerValidationServer

TransactionServerManager/TransactionServerManagerCase1:
1) magellan.cs.iupui.edu:9000/ TransactionServerManager
2) raleigh.cs.iupui.edu:9000/ TransactionServerManager
3) columnbus.cs.iupui.edu:9000/ TransactionServerManager

EconomicTransactionServer/EconomicTransactionServerCase1:
1) magellan.cs.iupui.edu:9000/ EconomicTransactionServer
2) raleigh.cs.iupui.edu:9000/ EconomicTransactionServer
3) columnbus.cs.iupui.edu:9000/ EconomicTransactionServer

The concrete components found above are implemented in Java RMI, except *http://134.68.140.142:9000/CashierValidationServer* (shown in the italic font in the above list) which was implemented in .NET. In this experiment, the following concrete components including the component in .NET were selected to generate a bank system:

magellan.cs.iupui.edu:9000/ATM
magellan.cs.iupui.edu:9000/ CashierTerminal
http://134.68.140.142:9000/CashierValidationServer
raleigh.cs.iupui.edu:9000/ CustomerValidationServer
columnbus.cs.iupui.edu:9000/ TransactionServerManager
columnbus.cs.iupui.edu:9000/ EconomicTransactionServer

The dynamic component QoS for each component was tested and compared with its advertised values. The deviations between the values were within 10%. The following is a typical testing result for *magellan.cs.iupui.edu:9000/ CashierTerminal:*

Function Name/QoS Parameter:	Advertised QoS	Dynamic QoS	Deviation
transferMoney/throughput (operations/second):	3198.47	3307.86	109.39
transferMoney/endToEndDelay (microsecond):	312.65	302.31	-10.34
depositMoney/throughput (operations/second):	3303.34	3417.12	113.78
depositMoney/endToEndDelay (microsecond):	302.72	292.64	-10.08
withdrawMoney/throughput (operations/second):	3513.33	3634.34	121.01
withdrawMoney/endToEndDelay (microsecond):	284.63	275.15	-9.48

Since the selected concrete components were heterogeneous, the *SystemGenerator* determined that the following adapter type (defined in Chapter 6) was required:

Bridge Type: Java RMI - .NET
Preprocessing Component: CashierTerminal/CashierTerminalCase1
Postprocessing Component: CashierValidationServer/CashierValidationServerCase1
Preprocessing Component Model: Java RMI
Postprocessing Component Model: .NET

In the next step, the *SystemGenerator* acquired the necessary adapter via the URDS. The following adapter was found: *134.68.140.142:2400/CashierValidationServerAdaper.* Then, the USGI statically validated the possible system by deriving the predicted system QoS from the selected concrete components based on the QoS composition rules, and compared the values with the expected system QoS. The result is listed below:

QoS Parameter:	Expected System QoS	Predicted System QoS	Deviation
throughput (operations/second):	700.00	1524.88	824.88
endToEndDelay (microsecond):	1500.00	655.79	-844.21

As shown above, the predicted system QoS met the expected system QoS. Thus, the USGI configured the system with the selected concrete components and the required adapter. After successful system assembly, the USGI validated the integrated system dynamically for the real system QoS. The result is shown below:

QoS Parameter:	Expected System QoS	Predicted System QoS	Deviation
throughput (operations/second):	700.00	1186.00	486.00
endToEndDelay (microsecond):	1500.00	843.00	-657.00

The above dynamic system QoS testing result demonstrated that the integrated system met the expected system QoS. Thus the ordering requirements were fulfilled and the system was ready to be deployed.

There are many possible combinations to integrate a bank system from the concrete components found above. Some of these combinations may not generate systems that can meet the ordering requirements (the system QoS requirements). These combinations are eliminated and the best system is returned from the rest of feasible combinations during the automatic system generation, which has not been implemented yet in the prototype.

7.3.4.2 Ordering Scheme with Natural-language-like Ordering Language

The experiment for this ordering scheme was done with the following natural-language-like order, “Generate a bank system with 1 ATM and 1 cashier terminal. The turn around time should be less than 1500 microseconds, and the throughput must be greater than 700 operations/second”. These system requirements are compatible with the one used in the previous section. When placing the order, this natural-language-like statement was processed by a natural language processor into structured ordering requirements which ware sent to the *OrderProcessor*. The *OrderProcessor* returned the same system specification as the one listed in the previous section. The rest of the system generation process is exactly the same as the ordering scheme with the tabular ordering language and is not repeated here.

This chapter describes in detail the design and implementation of a USGI prototype using Java technology. The prototype implementation demonstrates the proper design of the USGI architecture and various algorithms associated with the USGI. The prototype implementation also demonstrates the fulfillment of the objectives outlined in Chapter 1 for the USGPF. The next chapter will conclude this thesis with future work and a summary.

8. CONCLUSION

This thesis presented the UniFrame System-Level Generative Programming Framework (USGPF) for the purpose of automatic generation of DCS from DCS families by seamlessly integrating heterogeneous geographically dispersed software components. Section 8.1 presents an overview of the features of the USGPF followed by an overview of the contributions of this work. Section 8.2 presents the possible future enhancements to the USGPF. Section 8.3 concludes this thesis with a summation.

8.1 Outcome of the Study

The software solutions for the future DCS will require automatic or semi-automatic integration of software components, while abiding by the QoS constraints advertised by each component and the QoS requirements of the system. This thesis describes the system-level generative programming of the UniFrame Approach that allows an effective and efficient assembly of heterogeneous and distributed software components to create a DCS from a family of DCS specifications. The result of using the UniFrame and the associated tools (such as the USGPF) leads to the automation of DCS production while meeting both the functional and non-functional requirements of the DCS. The USGPF and its effectiveness are demonstrated via a comprehensive banking domain example.

There are many features of the USGPF proposed in this thesis. These are:

- The USGPF has built-in QoS support.
- The UGDM captures the common and variable properties of a DCS family, such as the component interactions, the communication patterns, and the QoS composition and decomposition models.

- The UGDP is a use-case driven, architecture-centric, iterative and incremental process.
- In the USGI, the application engineering process is guarded by the QoS in order to create QoS-aware DCS.
- A dynamic testing of the component QoS ensures the component meets its advertised component QoS.
- Double validations are designed to ensure that the QoS requirements are met during the system generation. The double validations include static and dynamic system QoS validations.
- A general-purpose system generation framework separates the concerns of the system generation logics from the domain dependent knowledge. The separation of the UGDM from the system generator and the separation of the concrete components from the system generator by the URDS are the key designs to achieve this feature. This feature allows more flexibility and maintainability.

The contributions of this thesis are:

- Definition of the UniFrame Generative Domain Model (UGDM). The UGDM has an inherent consideration of the QoS requirements to assist the need of developing QoS-aware DCS.
- Extension and enhancement of the work by [VAN02, VAR02] to create of the UniFrame Domain Specific Language (UDSL) to document various models in the UGDM in an informal fashion.
- Creation of the UniFrame UGDM Development Process (UGDP) to formulate a UGDM in assisting the development of a DCS family. The UGDP has a built-in support to integrate QoS into the UGDM.
- Development of a platform independent UniFrame System Generation Infrastructure (USGI) for efficiently generating QoS-aware DCS by seamlessly integrating heterogeneous distributed software components.
- Implementation of a prototype for the USGI based on the J2EE™ model.
- Validation of the USGPF by a detailed case study.

8.2 Future Work

Several future extensions to this research on the USGPF can be done. A few of these are discussed below.

8.2.1 Future Work on the UGDM

Following enhancements to the UGDM are possible in the future:

- The evolution of the UDSL to be more comprehensive. For example, developing a set of UDSL expressions to describe the event grammars [AUG95, AUG97] to dynamically measure and validate the QoS parameters in the UniFrame.
- The formalization of the UGDM representation. Currently the UGDM is documented informally using the UDSL developed in this work. This UGDM representation is then transformed into the XML format. Future research work in this front will include the formal representation of the UGDM using TLG [BRY00, BRY02, BRY02a].

8.2.2 Future Work on the UGDP

Currently the UGDP is not automated. The only tools created are a set of XML parsers which can automatically input the UGDM from the XML format into an Oracle database. The future work on the UGDP includes both the refinement and the automation of the process.

The Generic Modeling Environment (GME) [GME] developed by Vanderbilt University is an excellent tool that can be used to assist the UGDP. The GME is a configurable toolkit for creating domain-specific modeling and program synthesis environments. The configuration is accomplished through meta-models which specify the modeling paradigm (modeling language) of the application domain. The modeling paradigm contains all the syntactic, semantic, and presentation information regarding the domain; which concepts will be used to construct models, what relationships may exist among those concepts, how the concepts may be organized and viewed by the modeler, and rules governing the construction of models. The modeling paradigm defines the family of models that can be created using the resultant modeling environment.

There are several steps involved in using the GME to model the UGDM. First, a meta-model is created to describe the possible relationship (such as *or* and *alternative* of the feature description, see Chapter 4) and constraints (such as *require*, *reject* and *mutual_require* of the diagram constraint, see Chapter 4) that are used by the UGDM. Second, an interpreter is written to translate any specific models created based on the meta-model into the XML format defined for the models in the UGDM. The interpreter implements the normalization and expansion rules and the constraint checking. It is written in C++ with Visual Studio 6.0. Finally, with the availability of a meta-model and an associated interpreter, a specific model for the banking domain example is created and interpreted into XML.

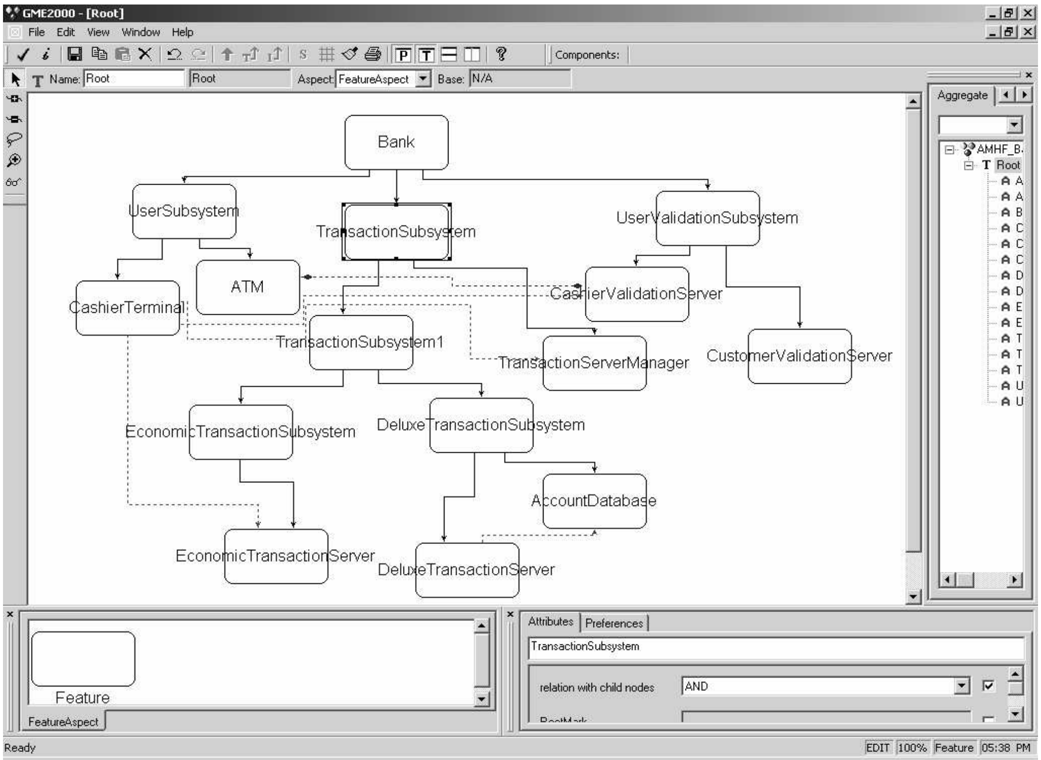


Figure 8.1 Example of Generic Modeling Environment

Table 8.1 AMDNF in the XML format Created by the GME Interpreter


```
<?xml version='1.0' encoding="utf-8" ?>
<!-- generated automatically by feature metamodel interpreter
 @2003/3/21,16:49-->
  <architecture_component  containment="XOR"  selfIsMandatory="TRUE">
    <system_name>Bank</system_name>
      <case  containment="AND"  selfIsMandatory="FALSE">
            <component  selfIsMandatory="TRUE">ATM</component>
            <component  selfIsMandatory="TRUE">CustomerValidationServer</component>
            <component  selfIsMandatory="TRUE">TransactionServerManager</component>
            <component  selfIsMandatory="TRUE">CashierTerminal</component>
            <component  selfIsMandatory="TRUE">CashierValidationServer</component>
            <component  selfIsMandatory="TRUE">EconomicTransactionServer</component>
      </case>
      <case  containment="AND"  selfIsMandatory="FALSE">
            <component  selfIsMandatory="TRUE">ATM</component>
            <component  selfIsMandatory="TRUE">CustomerValidationServer</component>
            <component  selfIsMandatory="TRUE">CashierTerminal</component>
            <component  selfIsMandatory="TRUE">CashierValidationServer</component>
            <component  selfIsMandatory="TRUE">TransactionServerManager</component>
            <component  selfIsMandatory="TRUE">DeluxeTransactionServer</component>
            <component  selfIsMandatory="TRUE">AccountDatabase</component>
      </case>
      <case  containment="AND"  selfIsMandatory="FALSE">
            <component  selfIsMandatory="TRUE">CashierTerminal</component>
            <component  selfIsMandatory="TRUE">CashierValidationServer</component>
            <component  selfIsMandatory="TRUE">TransactionServerManager</component>
            <component  selfIsMandatory="TRUE">EconomicTransactionServer</component>
      </case>
      <case  containment="AND"  selfIsMandatory="FALSE">
            <component  selfIsMandatory="TRUE">CashierTerminal</component>
            <component  selfIsMandatory="TRUE">CashierValidationServer</component>
            <component  selfIsMandatory="TRUE">TransactionServerManager</component>
            <component  selfIsMandatory="TRUE">DeluxeTransactionServer</component>
            <component  selfIsMandatory="TRUE">AccountDatabase</component>
      </case>
  </architecture_component>
```


Here is a brief example that indicates the use of the GME in the UGDP. This example has been developed as a joint effort with Fei Cao [CAO03], another researcher in the UniFrame project. The example is about modeling the Architecture Model in Hierarchical Form (AMHF) of the UGDM for the banking domain example, then deriving the Architecture Model in Disjunctive Normal Form (AMDNF) at component level in the XML format. Figure 8.1 shows the AMHF at the component level. At the lower-right corner of Figure 8.1 shows the interface to specify the relationship of the node under focus (*TransactionSubsystem* in the figure) with its child-nodes in the environment. The dashed lines in the figure denote the various kinds of dependencies or constraints to be enforced between feature nodes. In this example, the XML created by the GME (shown in Table 8.1) is completely compatible with the format needed by the corresponding XML parser (shown in Appendix H) except that the case name for each case is not assigned although the interpreter can be written to do so easily. For details of how to use the GME to do the modeling, see the tutorial of the GME [GME].

8.2.3 Future Work on the USGI Architecture

Currently, the USGI architecture does not provide any modules to assist a customized development of a DCS and there is no support for integrating user-supplied proprietary components. These features might be necessary as the UGDM might not capture all the possible details in a domain, and as markets usually change constantly over time, the new requirements may also come up. So, addition of these features is another avenue for the future work.

8.2.4 Future Work on the USGI Prototype

The future work on the USGI prototype involves more comprehensive and complete implementation of the USGI architecture. Many implementational strategies can be applied to enhance the prototype at an application level and make the implementation more scalable, fault tolerance, maintainable, interoperable and secure. Below a summary of these strategies is provided.

8.2.4.1 Workload Management

The prototype implementation of the USGI design supports various services like *Order Processor*, *System Generator*, *UGDMKB Generator*, *Wrapper and Glue Generator*, and *URDS*. It is desirable that these services be able to handle a large number of requests simultaneously without noticeable degradation in the performance. It is also desirable that these services be available for most of the time. One way to achieve these objectives is to deploy the services in a runtime environment with Workload Management (WLM). An example of such a runtime environment is the IBM's

WebSphere Application Server 4.0 [IBM02]. WLM improves the performance, scalability and reliability of an application by spreading multiple requests for a service over resources that can accomplish the task. WLM distributes incoming requests across application servers that contain identical copies of the service.

8.2.4.2 Interoperability

In the current USGI prototype implementation, the service components are implemented as Java-RMI based services which communicate with each other via JRMP. An alternative is to implement these services as Enterprise JavaBeans. Enterprise JavaBeans are deployed in the EJB container and they communicate with each other via RMI-IIOP. Another alternative is to implement these services as SOAP-based services like Web Services, in which the services communicate with each other via SOAP. These protocols promote a greater interoperability than JRMP. Using SOAP for inter-component communication removes the tight coupling that currently exists between the service components. In addition, SOAP is a firewall-friendly protocol, thus, it can remove the restrictions in the current USGI prototype implementation that the services have to be located within the same subnet.

8.2.4.3 Asynchronous Communication

All the communication in the current USGI prototype implementation is synchronous. One service waits on other services to return results. Making these communications asynchronous and using remote event notification will allow a greater flexibility, a higher system throughput and a better response time. The use of asynchronous messaging allows the development of loosely-connected systems. These systems are typically more resilient in the event of failures, and more easily extensible as new applications are developed. Additionally, messaging provides an effective means of transmitting events between applications. Asynchronous messaging can be incorporated into the implementation using Java Message Service (JMS). The service components can be implemented as EJB components which integrate with JMS, thus allowing the enterprise beans to participate fully in loosely connected systems. The EJB service components can then asynchronously notify other components of the occurrence of events. Remote event notification features will allow the *Model* to notify the *Controller* of changes in events in the model, which can be rendered as the *View* in the system.

8.2.4.4 System Security

In the current USGI prototype implementation, users of the system are not authenticated and there is no notion of access levels that users may have. However, this would be a desired feature, as service providers may not wish to advertise their services to unauthorized users, or depending on the privileges the users possess, they may allow access to only a certain set of functionality as opposed to others. The authentication aspect for users can be handled through a form-based user id/password scheme. For supporting users with different profiles, structuring the service components as EJB components allows the implementation to leverage the role, based security services offered by the EJB architecture.

8.3 Summary

This thesis has presented the UniFrame System-Level Generative Programming Framework (USGPF), which facilitates a semi-automatic/automatic generation of a distributed computing system from a system family. The UGDM defines various models to capture the common and variable properties of a family of distributed computing systems. The USGPF has built-in characteristics of the QoS to assist creating QoS-aware distributed computing systems. The USGPF coupled with the UniFrame Approach presents a promising solution for creating DCS by integrating geographically scattered, heterogeneous software components. The results of applying this approach in the semi-automatic construction of simple DCS from a banking domain are promising and demonstrate the effectiveness of this research.

APPENDICES

APPENDICES

APPENDIX A: The Normalization Rules and Expansion Rules for Feature Description

This appendix contains the normalization rules (see Table A.1) and expansion rules (see Table A.2) for feature description and correspondent brief description of the rules. The material comes from van Deursen [van02].

Table A.1 Normalization Rules for Feature Description

Normalization Rules	
Rules:	
[N1] Fs, F, Fs', F?, Fs''	= Fs, F, Fs', Fs''
[N2] Fs, F, Fs', F, Fs''	= Fs, F, Fs', Fs''
[N3] F*??*	= F*?*
[N4] *all*(F)	= F
[N5] *all*(Fs, *all*(Ft), Fs')	= *all*(Fs, Ft, Fs')
[N6] *one-of*(F)	= F
[N7] *one-of*(Fs, *one-of*(Ft), Fs')	= *one-of*(Fs, Ft, Fs')
[N8] *one-of*(Fs, F*?*, Fs')	= *one-of*(Fs, F, Fs')*?*
[N9] *more-of*(F)	= F
[N10] *more-of*(Fs, *more-of*(Ft), Fs')	= *more-of*(Fs, Ft, Fs')
[N11] *more-of*(Fs, F*?*, Fs')	= *more-of*(Fs, F, Fs')*?*
[N12] *default* = A	= A

Table A.1 presents the normalization rules for feature description. Here are the brief descriptions of each rule.

- N1 combines *mandatory* and *optional* features in a list.
- N2 removes duplicates in a list.
- N3 joins duplicate *optional*s.

- N4-N5 normalize special cases of *all*. Nested *all*s are flattened.
- N6-N7 normalize special cases of one-of. Nested *one-of*s are flattened.
- N8 transforms a *one-of* containing one *optional* feature into an *optional one-of*.
- N9-N10 normalize special cases of *more-of*. Nested *more-of*s are flattened.
- N11 transforms a *more-of* containing one *optional* feature into an *optional more-of*.
- N12 eliminates the *default* = annotation.

Table A.2 Expansion Rules for Feature Description

Expansion Rules	
Rules:	
[E1] *all*(Fs, F*?*, Ft)	= *one-of*(*all*(Fs, F, Ft), *all*(Fs, Ft))
[E2] *all*(Ft, F*?*, Fs)	= *one-of*(*all*(Ft, F, F), *all*(Ft, Fs))
[E3] *all*(Fs, *one-of*(F, Ft), Fs')	= *one-of*(*all*(Fs, F, Fs'), *all*(Fs, *one-of*(Ft), Fs'))
[E4] *all*(Fs, *more-of*(F, Ft), Fs')	= *one-of*(*all*(Fs, F, Fs'), *all*(Fs, F, *more-of*(Ft),Fs'), *all*(Fs, *more-of*(Ft), Fs'))

Table A.2 presents the expansion rules for feature description. Here are the brief descriptions of each rule.

- E1, E2 translates an *all* containing an *optional* feature expression in two cases: one with and one without the feature.
- E3 translates an *all* containing a *one-of* in two cases: one with the first *alternative* and one with the *one-of* with the first *alternative* removed.
- E4 translates an *all* containing a *more-of* into three cases: one with the first *alternative*, one with the first *alternative* and the remaining *more-of*, and one with only the remaining *more-of*.

APPENDIX B: Component Diagrams in the Banking Domain Example

This appendix consists of component diagrams for all cases of bank systems identified by the architecture model in disjunctive normal form at component level for the banking domain example. Totally there are four cases.

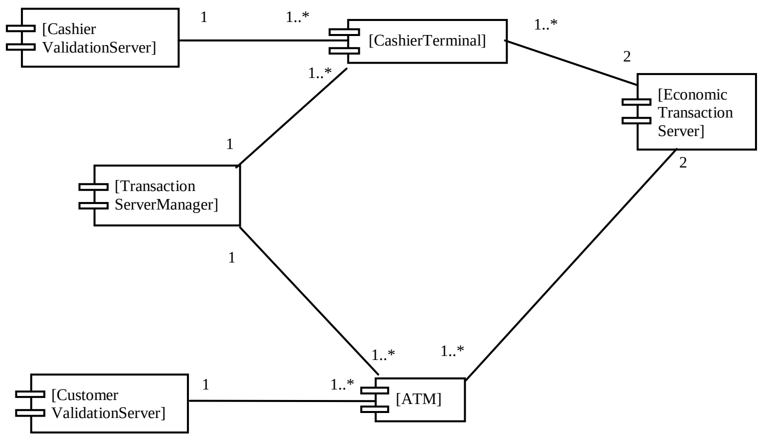


Figure B.1 Component Diagram of *BankCase1* for the Banking Domain Example

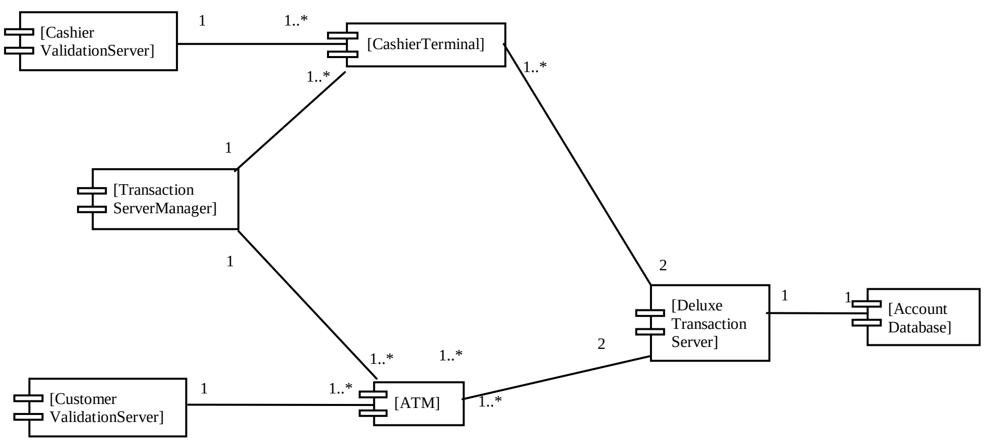


Figure B.2 Component Diagram of *BankCase2* for the Banking Domain Example

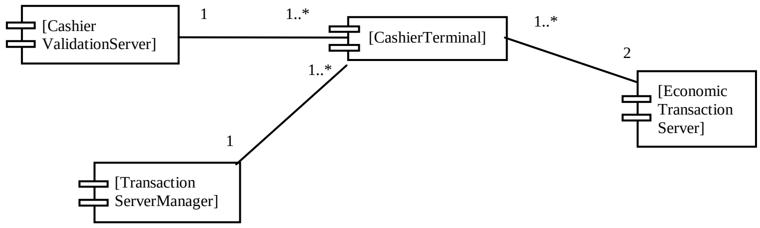


Figure B.3 Component Diagram of *BankCase3*
for the Banking Domain Example

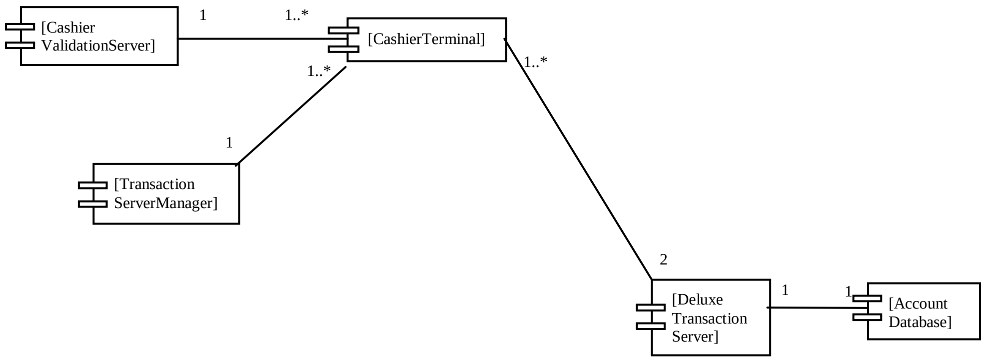


Figure B.4 Component Diagram of *BankCase4*
for the Banking Domain Example

APPENDIX C: Sequence Diagrams in the Banking Domain Example

This appendix consists of the sequence diagrams for all the use cases in the banking domain example.

- *ValidateUsers*

  This use case has two cases, *ValidateUsers_Cashier* and *ValidateUsers_Customer*. The first case is to validate users who are cashiers. The second case is to validate users who are customers. The sequence diagrams for these two cases are shown in Figure C.1 and C.2 respectively.

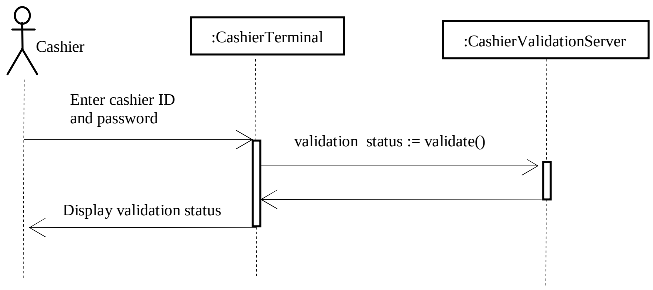


Figure C.1 Sequence Diagram of *ValidateUsers_Cashier*

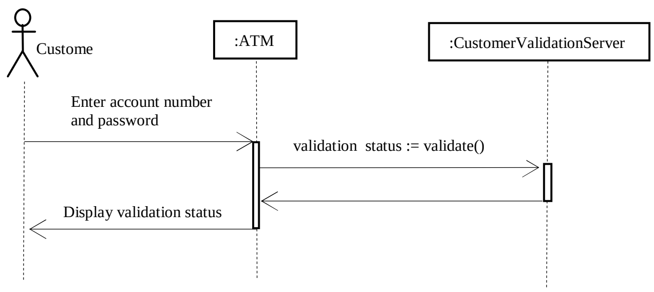


Figure C.2 Sequence Diagram of *ValidateUsers_Customer*

- *Login-exitAccount*

  This use case has two cases, *Login-exitAccount_Cashier* and *Login-exitAccount_Customer*. The first case is to login and exit when users are cashiers. The second case is to login and exit when users are customers. The sequence diagrams for these two cases are shown in Figure C.3 and C.4 respectively.

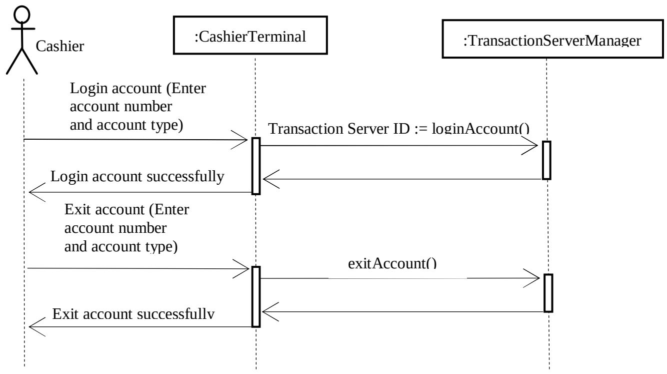


Figure C.3 Sequence Diagram of *Login-exitAccount_Cashier*

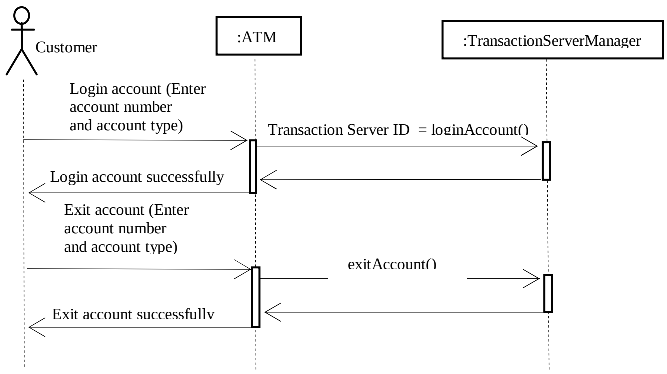


Figure C.4 Sequence Diagram of *Login-exitAccount_Customer*

- *DepositMoney*

  There are four cases in this use case. Figure C.5 illustrates the first case, in which the users are cashiers and the transaction subsystem consists of *EconomicTransactionServer*. Figure C.6 illustrates the second case, in which the users are cashiers and the transaction subsystem consists of *DeluxeTransactionServer* and *AccountDatabase*. Figure C.7 the third case, in which the users are customers and the transaction subsystem consists of *EconomicTransactionServer*. Figure C.8 illustrates the fourth case, in which the users are customers and the transaction subsystem consists of *DeluxeTransactionServer* and *AccountDatabase*.

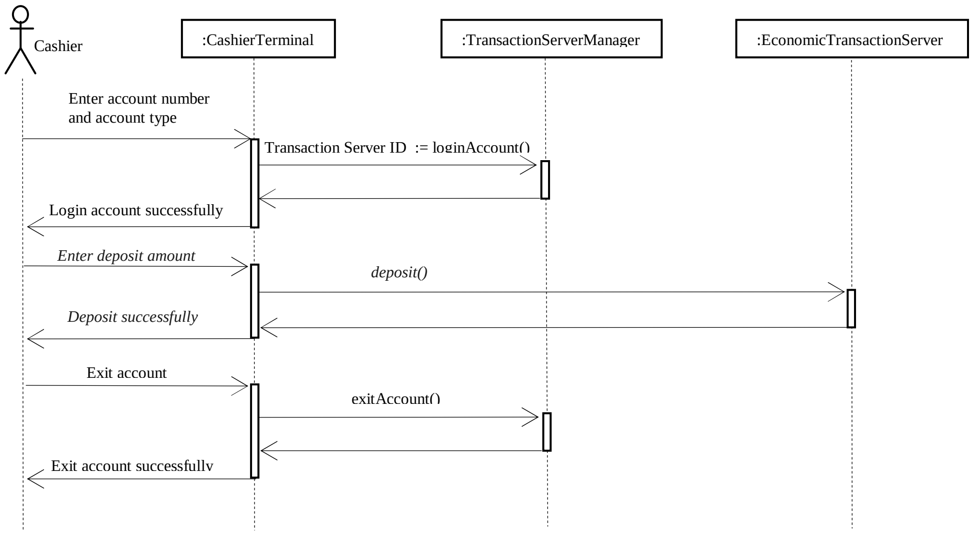


Figure C.5 Sequence Diagram of *DepositMoney* (Case 1)

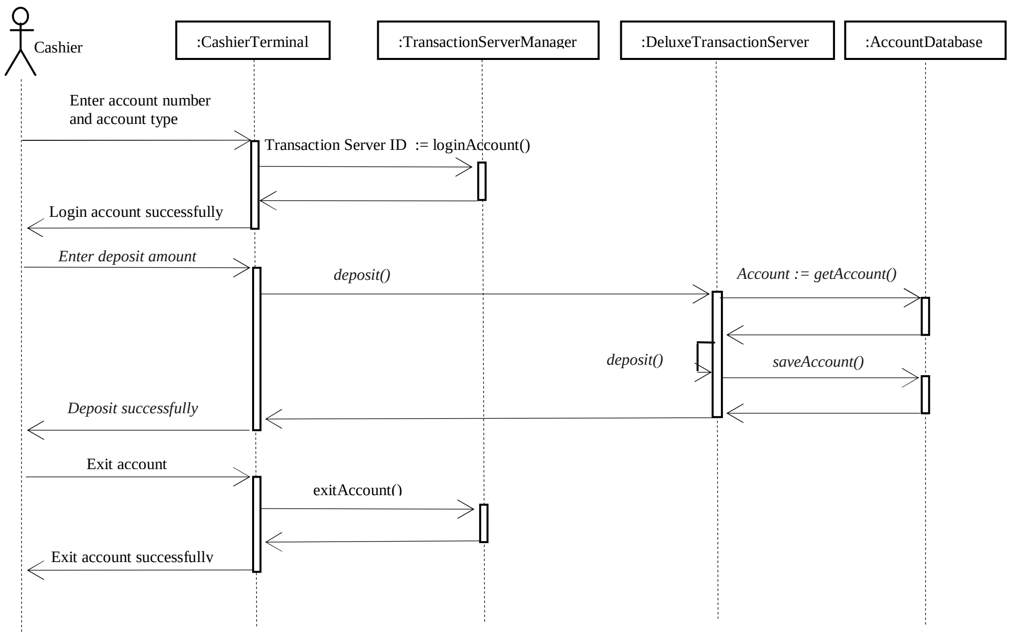


Figure C.6 Sequence Diagram of *DepositMoney* (case 2)

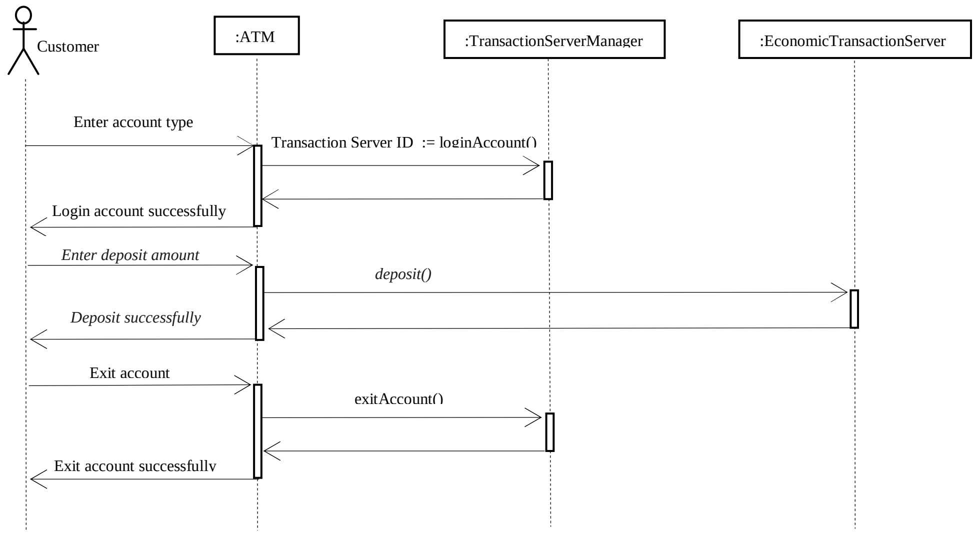


Figure C.7 Sequence Diagram of *DepositMoney* (case 3)

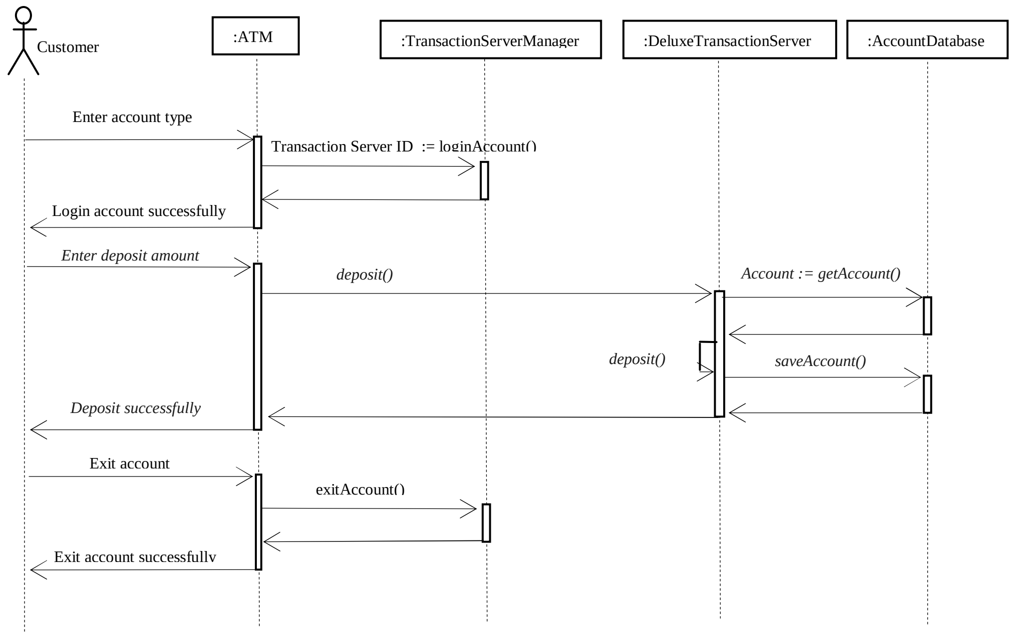


Figure C.8 Sequence Diagram of *DepositMoney* (case 4)

- *WithdrawMoney*

  There are four cases in this use case. Figure C.9 illustrates the first case, in which the users are cashiers and the transaction subsystem consists of

EconomicTransactionServer. Figure C.10 illustrates the second case, in which the users are cashiers and the transaction subsystem consists of *DeluxeTransactionServer* and *AccountDatabase*. Figure C.11 the third case, in which the users are customers and the transaction subsystem consists of *EconomicTransactionServer*. Figure C.12 illustrates the fourth case, in which the users are customers and the transaction subsystem consists of *DeluxeTransactionServer* and *AccountDatabase*.

- *TransferMoney*

  There are four cases in this use case. Figure C.13 illustrates the first case, in which the users are cashiers and the transaction subsystem consists of *EconomicTransactionServer*. Figure C.14 illustrates the second case, in which the users are cashiers and the transaction subsystem consists of *DeluxeTransactionServer* and *AccountDatabase*. Figure C.15 the third case, in which the users are customers and the transaction subsystem consists of *EconomicTransactionServer*. Figure C.16 illustrates the fourth case, in which the users are customers and the transaction subsystem consists of *DeluxeTransactionServer* and *AccountDatabase*.

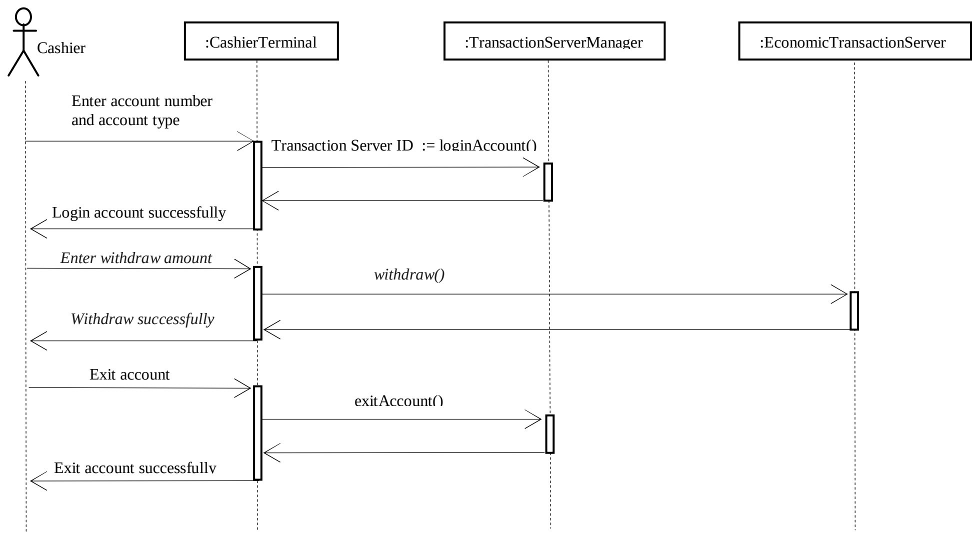


Figure C.9 Sequence Diagram of *WithdrawMoney* (Case 1)

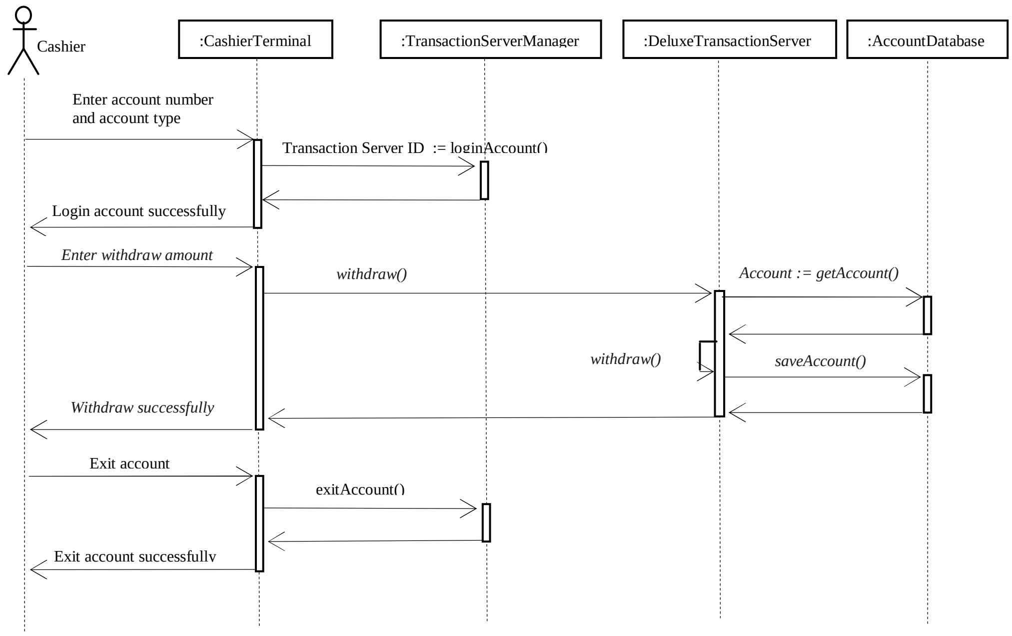


Figure C.10 Sequence Diagram of *WithdrawMoney* (case 2)

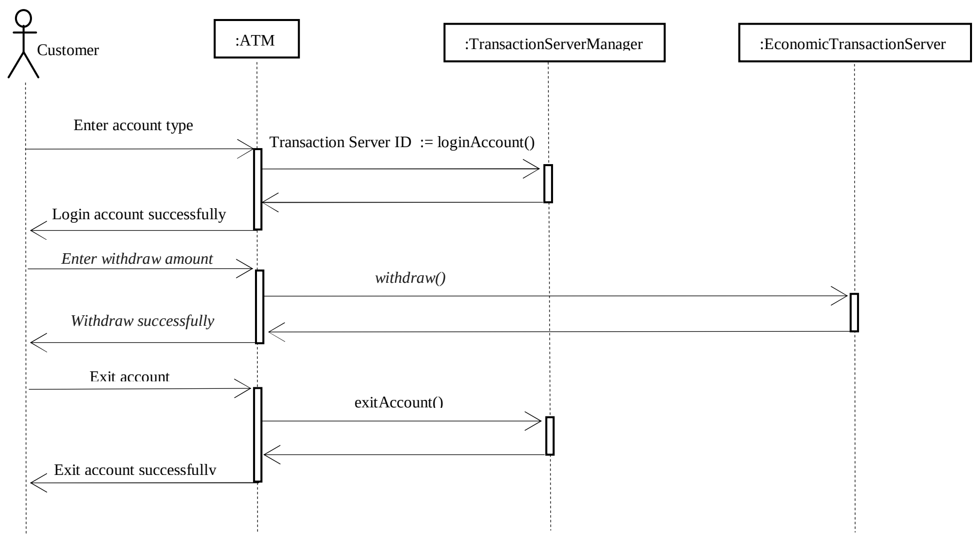


Figure C.11 Sequence Diagram of *WithdrawMoney* (case 3)

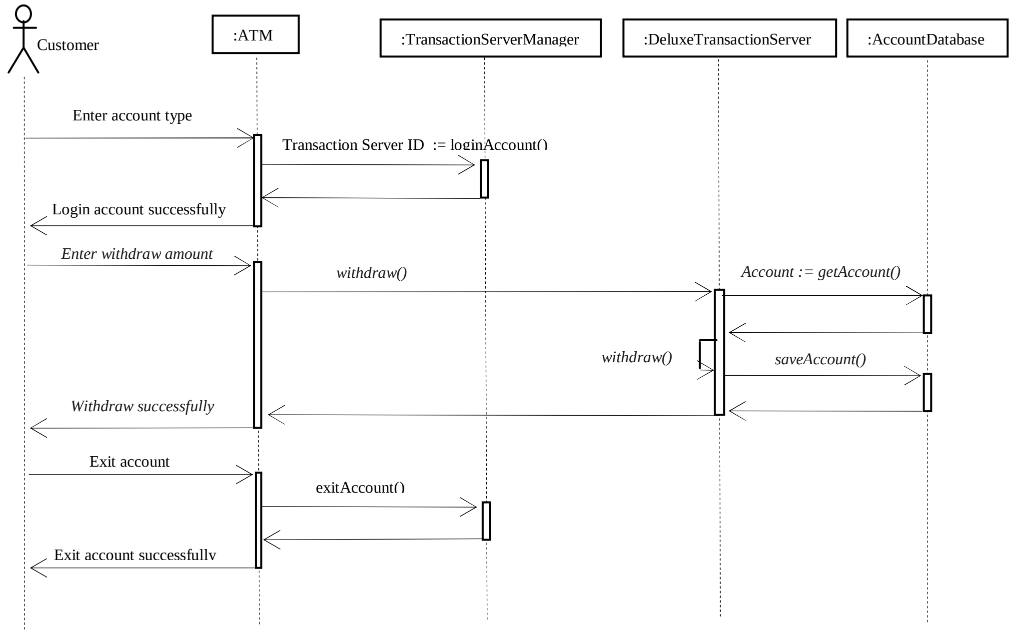


Figure C.12 Sequence Diagram of *WithdrawMoney* (case 4)

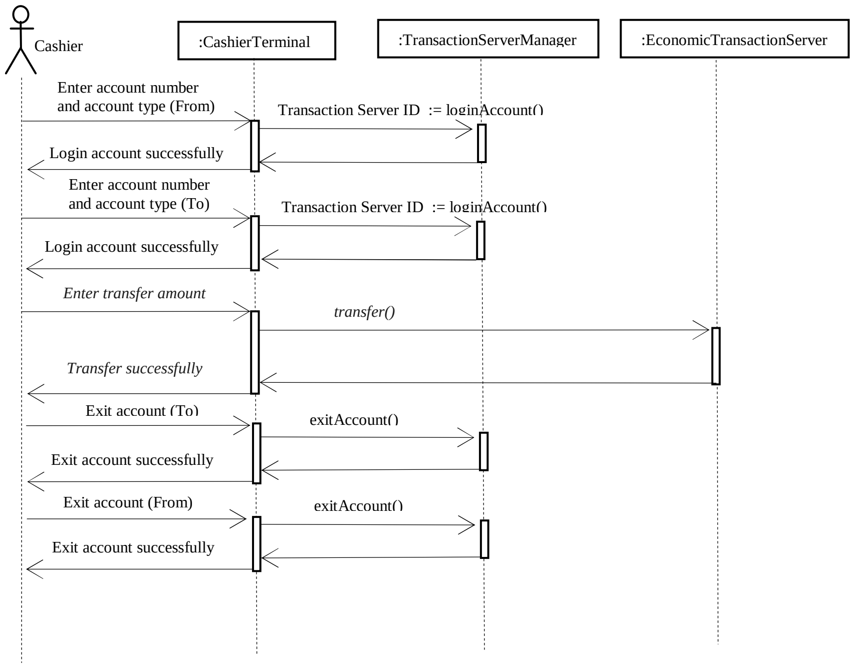


Figure C.13 Sequence Diagram of *TransferMoney* (case 1)

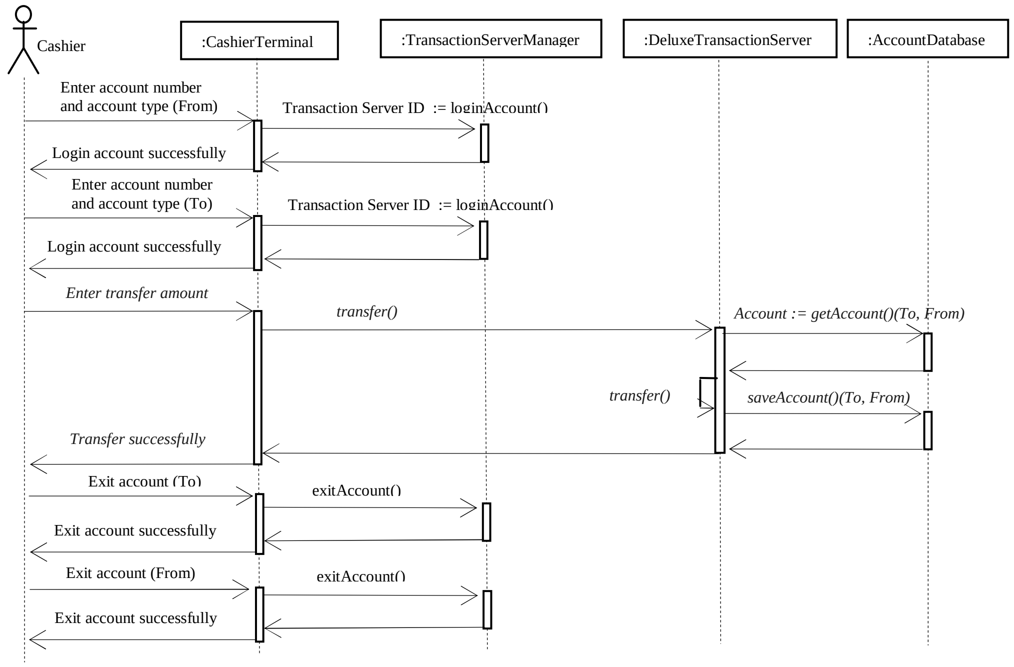


Figure C.14 Sequence Diagram of *TransferMoney* (case 2)

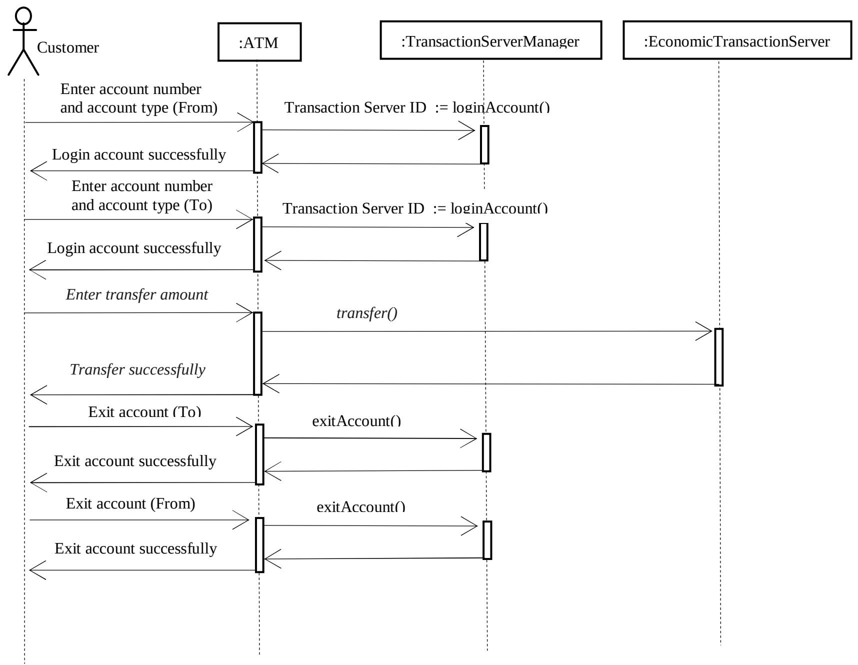


Figure C.15 Sequence Diagram of *TransferMoney* (case 3)

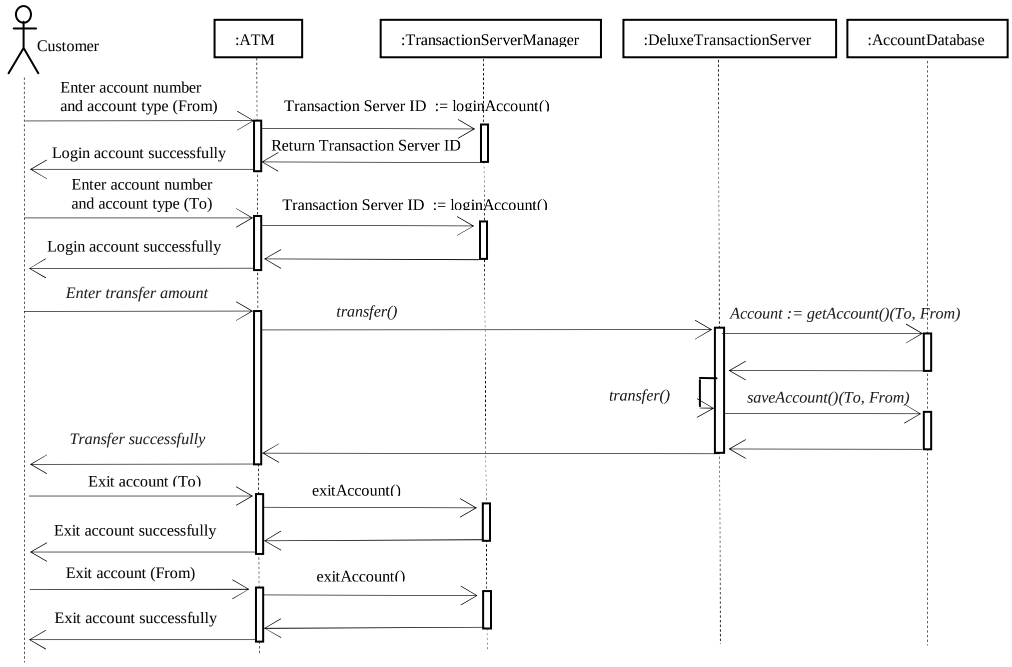


Figure C.16 Sequence Diagram of *TransferMoney* (case 4)

- *OpenAccount*

  The users of this use case are cashiers. There are two cases in this use case. Figure C.17 illustrates the first case, in which the transaction subsystem consists of *EconomicTransactionServer*. Figure C.18 illustrates the second case, in which the transaction subsystem consists of *DeluxeTransactionServer* and *AccountDatabase*.

- *CloseAccount*

  The users of this use case are cashiers. There are two cases in this use case. Figure C.19 illustrates the first case, in which the transaction subsystem consists of *EconomicTransactionServer*. Figure C.20 illustrates the second case, in which the transaction subsystem consists of *DeluxeTransactionServer* and *AccountDatabase*.

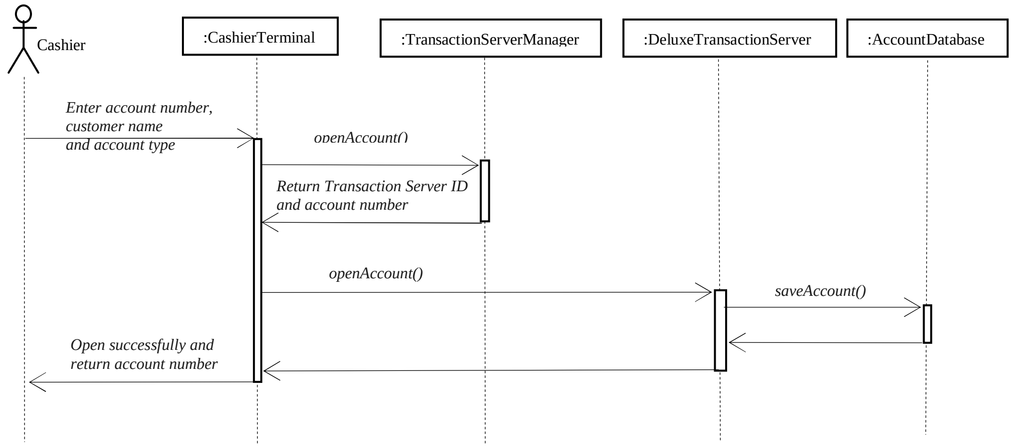


Figure C.17 Sequence Diagram of *OpenAccount* (case 1)

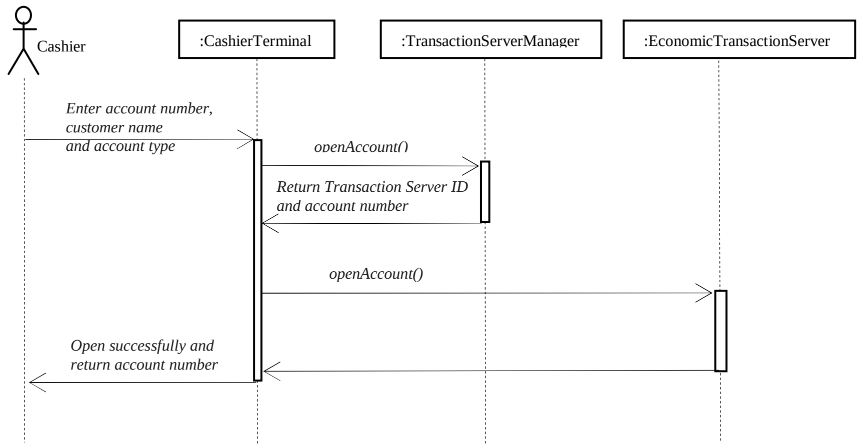


Figure C.18 Sequence Diagram of *OpenAccount* (case 2)

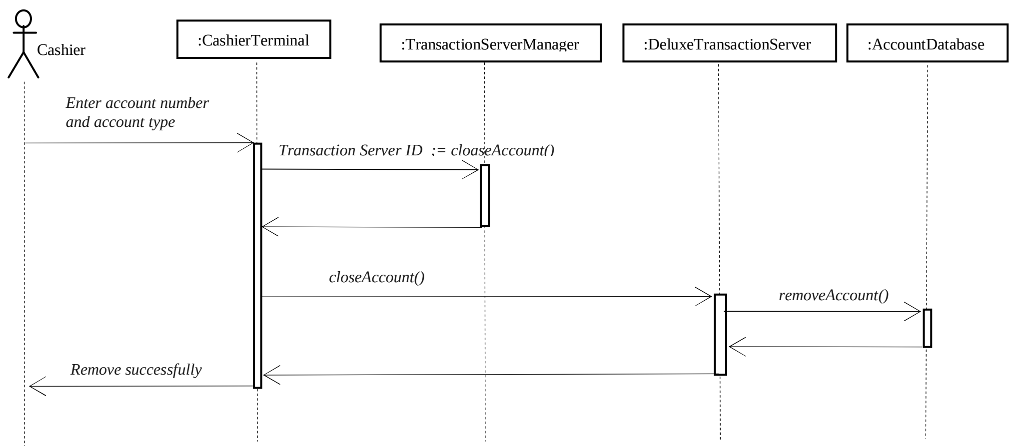


Figure C.19 Sequence Diagram of *CloseAccount* (case 1)

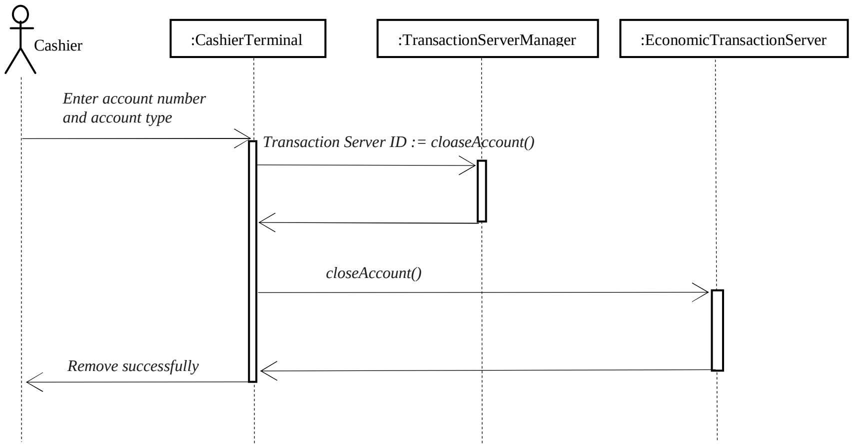


Figure C.20 Sequence Diagram of *CloseAccount* (case 2)

APPENDIX D: Function Summary of Abstract Components in the Banking Domain Example

This appendix documents function summaries for all the abstract components in the banking domain example. These include function summaries for *TransactionManager* (Table D.1), *CashierTerminal* (Table D.2), ATM (Table D.3), AccountDatabase (Table D.4), *DeluxeTransactionServer* (Table D.5), *EconomicTransactionServer* (Table D.6), *CashierValidationServer* (Table D.7), *CustomerValidationServer* (Table D.8).

Table D.1 Function Summary for *TransactionManager*

TransactionServerManager			
Actions	Inputs	Outputs	Communication Pattern
loginAccount()	Account Number, Account Type	Transaction Server ID	two-way-synchronous
exitAccount()	Account Number, Account Type	NONE	two-way-synchronous
openAccount()	Account Number, Account Type	Account Number, Account Type, Transaction Server ID	two-way-synchronous
closeAccount()	Account Number, Account Type	Transaction Server ID	two-way-synchronous

Table D.2 Function Summary for *CashierTerminal*

CashierTerminal			
Actions	Inputs	Outputs	Communication Pattern
validate()	Cashier ID, Password	Validation Status	two-way-synchronous
deposit()	Account Number, Account Type, Deposit Amount	NONE	two-way-synchronous
withdraw()	Account Number, Account Type, Withdraw Amount	NONE	two-way-synchronous
transfer()	Account Number (from), Account Type (from), Account Number (to), Account Type (to), Transfer Amount	NONE	two-way-synchronous
checkBalance()	Account Number, Account Type	NONE	two-way-synchronous
openAccount()	Customer Name, Account Number, Account Type	Account Number	two-way-synchronous
closeAccount()	Account Number, Account Type	NONE	two-way-synchronous

Table D.3 Function Summary for *ATM*

ATM			
Actions	Inputs	Outputs	Communication Pattern
validate()	Account Number, Password	Validation Status	two-way-synchronous
deposit()	Account Number, Account Type, Deposit Amount	NONE	two-way-synchronous
withdraw()	Account Number, Account Type, Withdraw Amount	NONE	two-way-synchronous
transfer()	Account Number (from), Account Type (from), Account Number (to), Account Type (to), Transfer Amount	NONE	two-way-synchronous
checkBalance()	Account Number, Account Type	NONE	two-way-synchronous

Table D.4 Function Summary for *AccountDatabase*

AccountDatabase			
Actions	Inputs	Outputs	Communication Pattern
getAccount()	Account Number, Account Type	Account	two-way-synchronous or two-way-asynchronous
saveAccount()	Account	NONE	two-way-synchronous or two-way-asynchronous
removeAccount()	Account Number, Account Type	NONE	two-way-synchronous or two-way-asynchronous

Table D.5 Function Summary for *DeluxeTransactionServer*

DeluxeTransactionServer			
Actions	Inputs	Outputs	Communication Pattern
deposit()	Account Number, Account Type, Deposit Amount	NONE	two-way-synchronous
withdraw()	Account Number, Account Type, Withdraw Amount	NONE	two-way-synchronous
transfer()	Account Number (from), Account Type (from), Account Number (to), Account Type (to), Transfer Amount	NONE	two-way-synchronous
checkBalance()	Account Number, Account Type	NONE	two-way-synchronous
openAccount()	Customer Name, Account Number, Account Type	Account Number	two-way-synchronous
closeAccount()	Account Number, Account Type	NONE	two-way-synchronous

Table D.6 Function Summary for *EconomicTransactionServer*

EconomicTransactionServer			
Actions	Inputs	Outputs	Communication Pattern
deposit()	Account Number, Account Type, Deposit Amount	NONE	two-way-synchronous
withdraw()	Account Number, Account Type, Withdraw Amount	NONE	two-way-synchronous
transfer()	Account Number (from), Account Type (from), Account Number (to), Account Type (to), Transfer Amount	NONE	two-way-synchronous
checkBalance()	Account Number, Account Type	NONE	two-way-synchronous
openAccount()	Customer Name, Account Number, Account Type	Account Number	two-way-synchronous
closeAccount()	Account Number, Account Type	NONE	two-way-synchronous

Table D.7 Function Summary for *CashierValidationServer*

CashierValidationServer			
Actions	Inputs	Outputs	Communication Pattern
validate()	Cashier ID, Password	Validation Status	two-way-synchronous

Table D.8 Function Summary for *CustomerValidationServer*

CustomerValidationServer			
Actions	Inputs	Outputs	Communication Pattern
validate()	Account number, Password	Validation Status	two-way-synchronous

APPENDIX E: Interface Model for the Banking Domain Example

This appendix consists of the interface model for the banking domain example. The model consists of interface descriptions for *ITransactionServerManager* (Table E.1), *IValidation* (Table E.2), *IAccountManagement* (Table E.3), *IAccountDatabase* (Table E.4) ande *ICustomerManagement* (Table E.5).

Table E.1 Interface Description for *IAccountDatabase*

IAccountDatabase

1. Syntax
Account getAccount(String accountNumber, int accountType);
Pre: values have been provided for the accountNumber and accountType.
Post: if the specified account exits, return the account; otherwise return NULL.
Invariant: accountNumber, accountType
Communication Pattern: cp2s or cp2a
Description: This function returns an account object as identified by the parameters. It returns null if the account specified does not exist.
void saveAccount(Account account);
Pre: account is valid
Post: the database has been updated appropriately.
Invariant: account
Communication Pattern: cp2s or cp2a
Description: This function updates the account if it already exists; otherwise it adds an entry in the database for this new account.
void removeAccount(String accountNumber, int accountType);
Pre: values have been provided for the account and accountType
Post: the account specified is removed and the database has been updated appropriately
Invariant: accountNumber, accountType
Communication Pattern: cp2s or cp2a
Description: This function removes the specified account if it exists; otherwise it does nothing.

2. Variation IAccountDatabase: one-of (IAccountDatabaseCase1, IAccountDatabaseCase2) IAccountDatabaseCase1: {cp2s} IAccountDatabaseCase2: {cp2a}
3. Default IAccountDatabase: IAccountDatabaseCase1

Table E.2 Interface *IValidation* for the banking domain Example

IValidation

1. Syntax

boolean validate(String id, String password);

    Pre: values have been provided for id and password.
    Post: return true if the id and password are valid; otherwise, return false.
    Invariant: id, password
    Communication Pattern: cp2s
    Description: This function validates a id/password pair.

2. Variation

IValidation: IValidationCase1
IValidationCase1: {cp2s}

Table E.3 Interface *IAccountManagement* for the banking domain Example

IAccountManagement

1. Syntax

void deposit(double amount, String acountNumber, int accountType);

    Pre: amount > 0 && account exists
    Post: if the account exists, account balance increased by amount
          otherwise throw BankingException("Account Not Exists")
    Invariant: account.balance >= 0
    Communication Pattern: cp2s
    Description: This function deposits the money into an account.

void withdraw(double amount, String accountNumber, int accountType);

    pre: amount > 0 && amount <= account.balance, account exits
    post : if the account exists, account balance decreased by amount
          otherwise throw BankingException("Account Not Exist")
    Invariant: account.balance > = 0;
    Communication Pattern: cp2s
    Description: This function withdraws money from an account.

void transfer(double amount, String accountNumberFrom, int accountTypeFrom,
                    String accountNumberTo, int accountTypeTo)

    pre: amount > 0 && amount <= account(from).balance, account exits
    post: account(to).balance increased by the amount
    Invariant: account(from).balance >= 0 && account(to).balance >= 0
    Communication Pattern: cp2s
    Description: This function transfers money from one account to another

double checkBalance(String accountNumber, int accountType);

    pre : account exits
    post : if the account exists, return the balance
          otherwise throw BankingException("Account Not Exist")
    Invariant: account.balance does not change
    Communication Pattern: cp2s
    Description: This function checks the balance of an account

2. Variation

IAccountManagement: IAccountManagementCase1
IAccountManagementCase1: {cp2s}

Table E.4 Interface Description for *ITransactionServerManger*

ITransactionServerManager

1. Syntax

String loginAccount (String accountNumber, int accountType);

 Pre: values have been provided for accountNumber and accountType.

 Post: If login successful, lock the account and return the account server addres; otherwise, return null.

 Invariant: accountNumber, accountType.

 Communication Pattern: cp2s

 Description: This function checks if the specified account exists. If the account exists and is unlocked, it locks the account and returns the transaction server address for the account; otherwise it returns null.

void exitAccount(String accountNumber, int accountType);

 Pre: values have been provided for accountNumber and accountType.

 Post: If the account is locked, unlock the account; otherwise, do nothing.

 Invariant: accountNumber, accountType

 Communication Pattern: cp2s

 Description: This function checks if the specified account exists. If the account exists and is locked, it unlocks the account; otherwise it does nothing.

AccountInfo openAccount (String accountNumber, int accountType);

 Pre: The account specified by the accountNumber and accountType does not exist. The value for accountType is either 1 or 2.

 Post: an account is opened.

 Invariant: accountNumber, accountType.

 Communication Pattern: cp2s

 Description: This function checks if the account identified by accountNumber and acountType exists on the transaction server manger. If the account does not exist, it creates this account and identifies the transaction server for manage this account. If the accountNumber is null, it assigns an account number. It returns the account number, account type and transaction server address in an AccountInfo object.

String closeAccount (String accountNumber, int accountType);

 pre: values have been provided for accountNumber and accountType.

 post: the specified account is closed

 Invariant: accountNumber and accountType.

 Communication Pattern: cp2s

 Description: This function removes the specified account from the transaction server manager for a customer if the account exists and returns the transaction server address so that the account can be removed from the database; otherwise it does nothing and returns null.

2. Variation

ITransactionServerManger: ITransactionServerMangerCase1

ITransactionServerMangerCase1: {cp2s}

Table E.5 Interface *ICustomerManagement* for the banking domain Example

ICustomerManagement
1. Syntax void openAccount (String customerName, String accountNumber, int accountType) throws BankingException; Pre: The account specified by the accountNumber and accountType does not exist. The value for accountType is either 1 or 2. Post: An account is opened Invariant: customerName, accountNumber, accountType. Communication Pattern: cp2s Description: This function creates an account for a customer if the account identified by accountNumber and accountType does not exist; otherwise it throws the exception that the BankingException with the message “The Account Already Exists”. If the accountType is invalid, it throws the BankingException with the message “Invalid Account Type”. void closeAccount(String accountNumber, int accountType); pre: account exists, the balance in the account is 0 post: if the account exists and the balance is 0, delete the account, if the account exists and the balance is not 0, throw BankingException(“Account Not Empty”), otherwise throw BankingException(“Account Already Exists”) Invariant: accountNumber, accountType. Communication Pattern: cp2s Description: This function closes an account. 2. Variation ICustomerManagement: ICustomerManagementCase1 ICustomerManagementCase1: {cp2s}

APPENDIX F: Abstract Component Model for the Banking Domain Example

This appendix consists of the abstract component model for the banking domain example. The model includes UMM Specifications for *AccountDatabaseCase1* (Table F.1 and Table F.2), *AccountDatabaseCase2* (Table F.3), *DeluxeTransactionServerCase1* (Table F.4), *DeluxeTransactionServerCase2* (Table F.5), *ATMCase1* (Table F.6), *CashierTerminalCase1* (Table F.7), *CusotmerValidationServerCase1* (Table F.8), *CashierValidationServerCase1* (Table F.9), *TransactionServerManagerCase1* (Table F.10), and *EconomicTransactionServerCase1* (Table F.11).

Table F.1 UMM Specification for *AccountDatabaseCase1*

Abstract Component**:** *AccountDatabaseCase1* 1. Component Name: *AccountDatabase* 2. Component Subcase: *AccountDatabaseCase1* 3. Domain Name: Banking 4. System Name: Bank 5. Informal Description: Provide an account database service. 6. Computational Attributes: 6.1 Inherent Attributes: 6.1.1 id: N/A 6.1.2 Version: version 1.0 6.1.3 Author: N/A 6.1.4 Date: N/A 6.1.5 Validity: N/A 6.1.6 Atomicity: Yes 6.1.7 Registration: N/A 6.1.8 Model: N/A 6.2 Functional Attributes: 6.2.1 Function description: Serve as an account database. 6.2.2 Algorithm: N/A 6.2.3 Complexity: N/A 6.2.4 Syntactic Contract 6.2.4.1 Provided Interface: *IAccountDatabaseCase1* 6.2.4.2 Required Interface: NONE 6.2.5 Technology: N/A 6.2.6 Expected Resources: N/A 6.2.7 Design Patterns: NONE 6.2.8 Known Usage: NONE 6.2.9 Alias: NONE 7. Cooperation Attributes 7.1 Preprocessing Collaborators: *DeluxeTransactionServerCase1* 7.2 Postprocessing Collaborators: NONE 8. Auxiliary Attributes: 8.1 Mobility: No 8.2 Security: *L0* 8.3 Fault tolerance: *L0* (Continued in Table F.2)

Table F.2 UMM Specification for *AccountDatabaseCase1*
(Continued from Table F.1)

(Continued from Table F.1)
- 9. Quality of Service
  - 9.1 QoS Metrics: *throughput*, *end-to-end delay*
  - 9.2 QoS Level: N/A
  - 9.3 Cost: N/A
  - 9.4 Quality Level: N/A

Table F.3 UMM Specification for *AccountDatabaseCase2*

Abstract Component: *AccountDatabaseCase2*
- 1. Component Name: *AccountDatabase*
- 2. Component Subcase: *AccountDatabaseCase2*
- 3. Domain Name: Banking
- 4. System Name: Bank
- 5. Informal Description: Provide an account database service.
- 6. Computational Attributes:
  - 6.1 Inherent Attributes:
    - 6.1.1 id: N/A
    - 6.1.2 Version: version 1.0
    - 6.1.3 Author: N/A
    - 6.1.4 Date: N/A
    - 6.1.5 Validity: N/A
    - 6.1.6 Atomicity: Yes
    - 6.1.7 Registration: N/A
    - 6.1.8 Model: N/A
  - 6.2 Functional Attributes:
    - 6.2.1 Function description: Serve as an account database.
    - 6.2.2 Algorithm: N/A
    - 6.2.3 Complexity: N/A
    - 6.2.4 Syntactic Contract
      - 6.2.4.1 Provided Interface: *IAccountDatabaseCase2*
      - 6.2.4.2 Required Interface: NONE
    - 6.2.5 Technology: N/A
    - 6.2.6 Expected Resources: N/A
    - 6.2.7 Design Patterns: NONE
    - 6.2.8 Known Usage: NONE
    - 6.2.9 Alias: NONE
- 7. Cooperation Attributes
  - 7.1 Preprocessing Collaborators: *DeluxeTransactionServerCase2*
  - 7.2 Postprocessing Collaborators: NONE
- 8. Auxiliary Attributes:
  - 8.1 Mobility: No
  - 8.2 Security: *L0*
  - 8.3 Fault tolerance: *L0*
- 9. Quality of Service
  - 9.1 QoS Metrics*: throughput*, *end-to-end delay*
  - 9.2 QoS Level: N/A
  - 9.3 Cost: N/A
  - 9.4 Quality Level: N/A

Table F.4 UMM Specification for *DeluxeTransactionServerCase1*

Abstract Component: *DeluxeTransactionServerCase1*

1. Component Name: *DeluxeTransactionServer*
2. Component Subcase: *DeluxeTransactionServerCase1*
3. Domain Name: Banking
4. System Name: Bank
5. Informal Description: Provide transaction service in banking.
6. Computational Attributes:
    - 6.1 Inherent Attributes:
        - 6.1.1 id: N/A
        - 6.1.2 Version: version 1.0
        - 6.1.3 Author: N/A
        - 6.1.4 Date: N/A
        - 6.1.5 Validity: N/A
        - 6.1.6 Atomicity: Yes
        - 6.1.7 Registration: N/A
        - 6.1.8 Model: N/A
    - 6.2 Functional Attributes:
        - 6.2.1 Function description: Act as transaction server in banking.
        - 6.2.2 Algorithm: N/A
        - 6.2.3 Complexity: N/A
        - 6.2.4 Syntactic Contract
            - 6.2.4.1 Provided Interface: *IAccountManagementCase1*, *ICustomerManagementCase1*
            - 6.2.4.2 Required Interface: *IAccountDatabaseCase1*
        - 6.2.5 Technology: N/A
        - 6.2.6 Expected Resources: N/A
        - 6.2.7 Design Patterns: NONE
        - 6.2.8 Known Usage: NONE
        - 6.2.9 Alias: NONE
7. Cooperation Attributes
    - 7.1 Preprocessing Collaborators: *CashierTerminalCase1*, *ATMCase1*
    - 7.2 Postprocessing Collaborators: *AccountDatabaseCase1*
8. Auxiliary Attributes:
    - 8.1 Mobility: No
    - 8.2 Security: *L0*
    - 8.3 Fault tolerance: *L0*
9. Quality of Service
    - 9.1 QoS Metrics: *throughput, end-to-end delay*
    - 9.2 QoS Level: N/A
    - 9.3 Cost: N/A
    - 9.4 Quality Level: N/A

Table F.5 UMM Specification for *DeluxeTransactionServerCase2*

Abstract Component: *DeluxeTransactionServerCase2*

1. Component Name: *DeluxeTransactionServer*
2. Component Subcase: *DeluxeTransactionServerCase2*
3. Domain Name: Banking
4. System Name: Bank
5. Informal Description: Provide transaction service in banking.
6. Computational Attributes:
    - 6.1 Inherent Attributes:
        - 6.1.1 id: N/A
        - 6.1.2 Version: version 1.0
        - 6.1.3 Author: N/A
        - 6.1.4 Date: N/A
        - 6.1.5 Validity: N/A
        - 6.1.6 Atomicity: Yes
        - 6.1.7 Registration: N/A
        - 6.1.8 Model: N/A
    - 6.2 Functional Attributes:
        - 6.2.1 Function description: Act as transaction server in banking.
        - 6.2.2 Algorithm: N/A
        - 6.2.3 Complexity: N/A
        - 6.2.4 Syntactic Contract
            - 6.2.4.1 Provided Interface: *IAccountManagementCase1*, *ICustomerManagementCase1*
            - 6.2.4.2 Required Interface: *IAccountDatabaseCase2*
        - 6.2.5 Technology: N/A
        - 6.2.6 Expected Resources: N/A
        - 6.2.7 Design Patterns: NONE
        - 6.2.8 Known Usage: NONE
        - 6.2.9 Alias: NONE
7. Cooperation Attributes
    - 7.1 Preprocessing Collaborators: *CashierTerminalCase1*, *ATMCase1*
    - 7.2 Postprocessing Collaborators: *AccountDatabaseCase2*
8. Auxiliary Attributes:
    - 8.1 Mobility: No
    - 8.2 Security: *L0*
    - 8.3 Fault tolerance: *L0*
9. Quality of Service
    - 9.1 QoS Metrics: *throughput*, *end-to-end delay*
    - 9.2 QoS Level: N/A
    - 9.3 Cost: N/A
    - 9.4 Quality Level: N/A

Table F.6 UMM Specification for *ATMCase1*

Abstract Component: *ATMCase1*

1. Component Name: *ATM*
2. Component Subcase: *ATMCase1*
3. Domain Name: Banking
4. System Name: Bank
5. Informal Description: Provide GUI for *ATM*.
6. Computational Attributes:
    - 6.1 Inherent Attributes:
        - 6.1.1 id: N/A
        - 6.1.2 Version: version 1.0
        - 6.1.3 Author: N/A
        - 6.1.4 Date: N/A
        - 6.1.5 Validity: N/A
        - 6.1.6 Atomicity: Yes
        - 6.1.7 Registration: N/A
        - 6.1.8 Model: N/A
    - 6.2 Functional Attributes:
        - 6.2.1 Function description: Act as *ATM*.
        - 6.2.2 Algorithm: N/A
        - 6.2.3 Complexity: N/A
        - 6.2.4 Syntactic Contract
            - 6.2.4.1 Provided Interface: *IAccountManagementCase1*, *IValidationCase1*, *IAccountManagementCase1*
            - 6.2.4.2 Required Interface: *IAccountManagementCase1*, *IValidationCase1*, *IAccountManagementCase1*, *ITransactionServerManagerCase1*
        - 6.2.5 Technology: N/A
        - 6.2.6 Expected Resources: N/A
        - 6.2.7 Design Patterns: NONE
        - 6.2.8 Known Usage: NONE
        - 6.2.9 Alias: NONE
7. Cooperation Attributes
    - 7.1 Preprocessing Collaborators: NONE
    - 7.2 Postprocessing Collaborators: *TansactionServerManagerCase1*, *CustomerValidationServerCase1*, *DeluxeTransactionServerCase1*, *DeluxeTransactionServerCase2*, *EconomicTransactionServerCase1*
8. Auxiliary Attributes:
    - 8.1 Mobility: No
    - 8.2 Security: *L0*
    - 8.3 Fault tolerance: *L0*
9. Quality of Service
    - 9.1 QoS Metrics: *throughput, end-to-end delay*
    - 9.2 QoS Level: N/A
    - 9.3 Cost: N/A
    - 9.4 Quality Level: N/A

Table F.7 UMM Specification for *CashierTerminalCase1*

Abstract Component**:** *CashierTerminalCase1*
1. Component Name: *CashierTerminal*
2. Component Subcase: *CashierTerminalCase1*
3. Domain Name: Banking
4. System Name: Bank
5. Informal Description: Provide GUI for cashiers.
6. Computational Attributes:
    6.1 Inherent Attributes:
        6.1.1 id: N/A
        6.1.2 Version: version 1.0
        6.1.3 Author: N/A
        6.1.4 Date: N/A
        6.1.5 Validity: N/A
        6.1.6 Atomicity: Yes
        6.1.7 Registration: N/A
        6.1.8 Model: N/A
    6.2 Functional Attributes:
        6.2.1 Function description: Act GUI terminal for cashiers.
        6.2.2 Algorithm: N/A
        6.2.3 Complexity: N/A
        6.2.4 Syntactic Contract
            6.2.4.1 Provided Interface: *IAccountManagementCase1*, *IValidationCase1*, *IAccountManagementCase1*
            6.2.4.2 Required Interface: *IAccountManagementCase1*, *IValidationCase1*, *IAccountManagementCase1*, *ITransactionServerManagerCase1*
        6.2.5 Technology: N/A
        6.2.6 Expected Resources: N/A
        6.2.7 Design Patterns: NONE
        6.2.8 Known Usage: NONE
        6.2.9 Alias: NONE
7. Cooperation Attributes
    7.1 Preprocessing Collaborators: NONE
    7.2 Postprocessing Collaborators: *TansactionServerManagerCase1*, *CustomerValidationServerCase1*, *DeluxeTransactionServerCase1*, *DeluxeTransactionServerCase2*, *EconomicTransactionServerCase1*
8. Auxiliary Attributes:
    8.1 Mobility: No
    8.2 Security: *L0*
    8.3 Fault tolerance: *L0*
9. Quality of Service
    9.1 QoS Metrics: *throughput*, *end-to-end delay*
    9.2 QoS Level: N/A
    9.3 Cost: N/A
    9.4 Quality Level: N/A

Table F.8 UMM Specification for *CustomerValidationServerCase1*

Abstract Component**:** *CustomerValidationServerCase1*

1. Component Name: *CustomerValidationServer*
2. Component Subcase: *CustomerValidationServerCase1*
3. Domain Name: Banking
4. System Name: Bank
5. Informal Description: Provide *ATM* validation service in banking.
6. Computational Attributes:
    - 6.1 Inherent Attributes:
        - 6.1.1 id: N/A
        - 6.1.2 Version: version 1.0
        - 6.1.3 Author: N/A
        - 6.1.4 Date: N/A
        - 6.1.5 Validity: N/A
        - 6.1.6 Atomicity: Yes
        - 6.1.7 Registration: N/A
        - 6.1.8 Model: N/A
    - 6.2 Functional Attributes:
        - 6.2.1 Function description: Act as validation server for *ATM*s in banking.
        - 6.2.2 Algorithm: N/A
        - 6.2.3 Complexity: N/A
        - 6.2.4 Syntactic Contract
            - 6.2.4.1 Provided Interface: *IValidationCase1*
            - 6.2.4.2 Required Interface: NONE
        - 6.2.5 Technology: N/A
        - 6.2.6 Expected Resources: N/A
        - 6.2.7 Design Patterns: NONE
        - 6.2.8 Known Usage: NONE
        - 6.2.9 Alias: NONE
7. Cooperation Attributes
    - 7.1 Preprocessing Collaborators: *ATMCase1*
    - 7.2 Postprocessing Collaborators: NONE
8. Auxiliary Attributes:
    - 8.1 Mobility: No
    - 8.2 Security: *L0*
    - 8.3 Fault tolerance: *L0*
9. Quality of Service
    - 9.1 QoS Metrics: *throughput*, *end-to-end delay*
    - 9.2 QoS Level: N/A
    - 9.3 Cost: N/A
    - 9.4 Quality Level: N/A

Table F.9 UMM Specification for *CashierValidationServerCase1*

Abstract Component: *CashierValidationServerCase1*
1. Component Name: *CashierValidationServer*
2. Component Subcase: *CashierValidationServerCase1*
3. Domain Name: Banking
4. System Name: Bank
5. Informal Description: Provide Cashier validation service in banking.
6. Computational Attributes:
    - 6.1 Inherent Attributes:
        - 6.1.1 id: N/A
        - 6.1.2 Version: version 1.0
        - 6.1.3 Author: N/A
        - 6.1.4 Date: N/A
        - 6.1.5 Validity: N/A
        - 6.1.6 Atomicity: Yes
        - 6.1.7 Registration: N/A
        - 6.1.8 Model: N/A
    - 6.2 Functional Attributes:
        - 6.2.1 Function description: Act as validation server for Cashiers in banking.
        - 6.2.2 Algorithm: N/A
        - 6.2.3 Complexity: N/A
        - 6.2.4 Syntactic Contract
            - 6.2.4.1 Provided Interface: *IValidationCase1*
            - 6.2.4.2 Required Interface: NONE
        - 6.2.5 Technology: N/A
        - 6.2.6 Expected Resources: N/A
        - 6.2.7 Design Patterns: NONE
        - 6.2.8 Known Usage: NONE
        - 6.2.9 Alias: NONE
7. Cooperation Attributes
    - 7.1 Preprocessing Collaborators: *CashierTerminalCase1*
    - 7.2 Postprocessing Collaborators: NONE
8. Auxiliary Attributes:
    - 8.1 Mobility: No
    - 8.2 Security: *L0*
    - 8.3 Fault tolerance: *L0*
9. Quality of Service
    - 9.1 QoS Metrics: *throughput*, *end-to-end delay*
    - 9.2 QoS Level: N/A
    - 9.3 Cost: N/A
    - 9.4 Quality Level: N/A

Table F.10 UMM Specification for *TransactionServerManagerCase1*

Abstract Component**:** *TransactionServerManagerCase1*

1. Component Name: *TransactionServerManager*
2. Component Subcase: *TransactionServerManagerCase1*
3. Domain Name: Banking
4. System Name: Bank
5. Informal Description: Provide transaction server management service in banking.
6. Computational Attributes:
   - 6.1 Inherent Attributes:
     - 6.1.1 id: N/A
     - 6.1.2 Version: version 1.0
     - 6.1.3 Author: N/A
     - 6.1.4 Date: N/A
     - 6.1.5 Validity: N/A
     - 6.1.6 Atomicity: Yes
     - 6.1.7 Registration: N/A
     - 6.1.8 Model: N/A
   - 6.2 Functional Attributes:
     - 6.2.1 Function description: Act as transaction server manager for ATMs in banking.
     - 6.2.2 Algorithm: N/A
     - 6.2.3 Complexity: N/A
     - 6.2.4 Syntactic Contract
       - 6.2.4.1 Provided Interface: *ITransactionServerManagerCase1*
       - 6.2.4.2 Required Interface: NONE
     - 6.2.5 Technology: N/A
     - 6.2.6 Expected Resources: N/A
     - 6.2.7 Design Patterns: NONE
     - 6.2.8 Known Usage: NONE
     - 6.2.9 Alias: NONE
7. Cooperation Attributes
   - 7.1 Preprocessing Collaborators: *CashierTerminalCase1*, *ATMCase1*
   - 7.2 Postprocessing Collaborators: NONE
8. Auxiliary Attributes:
   - 8.1 Mobility: No
   - 8.2 Security: *L0*
   - 8.3 Fault tolerance: *L0*
9. Quality of Service
   - 9.1 QoS Metrics: *throughput, end-to-end delay*
   - 9.2 QoS Level: N/A
   - 9.3 Cost: N/A
   - 9.4 Quality Level: N/A

Table F.11 UMM Specification for *EconomicTransactionServerCase1*

Abstract Component**:** *EconomicTransactionServerCase1*

1. Component Name: *EconomicTransactionServer*
2. Component Subcase: *EconomicTransactionServerCase1*
3. Domain Name: Banking
4. System Name: Bank
5. Informal Description: Provide transaction service in banking.
6. Computational Attributes:
    - 6.1 Inherent Attributes:
        - 6.1.1 id: N/A
        - 6.1.2 Version: version 1.0
        - 6.1.3 Author: N/A
        - 6.1.4 Date: N/A
        - 6.1.5 Validity: N/A
        - 6.1.6 Atomicity: Yes
        - 6.1.7 Registration: N/A
        - 6.1.8 Model: N/A
    - 6.2 Functional Attributes:
        - 6.2.1 Function description: Act as transaction server in banking.
        - 6.2.2 Algorithm: N/A
        - 6.2.3 Complexity: N/A
        - 6.2.4 Syntactic Contract
            - 6.2.4.1 Provided Interface: *IAccountManagementCase1*, *ICustomerManagementCase1*
            - 6.2.4.2 Required Interface: *IAccountDatabaseCase1*
        - 6.2.5 Technology: N/A
        - 6.2.6 Expected Resources: N/A
        - 6.2.7 Design Patterns: NONE
        - 6.2.8 Known Usage: NONE
        - 6.2.9 Alias: NONE
7. Cooperation Attributes
    - 7.1 Preprocessing Collaborators: *CashierTerminalCase1*, *ATMCase1*
    - 7.2 Postprocessing Collaborators: NONE
8. Auxiliary Attributes:
    - 8.1 Mobility: No
    - 8.2 Security: *L0*
    - 8.3 Fault tolerance: *L0*
9. Quality of Service
    - 9.1 QoS Metrics: throughput, end-to-end delay
    - 9.2 QoS Level: N/A
    - 9.3 Cost: N/A
    - 9.4 Quality Level: N/A

APPENDIX G: QoS Composition and Decomposition Rules for the Banking Domain Example

This appendix consists of the QoS composition and decomposition rules for the banking domain example derived from the QoS composition and decomposition meta-rules stated in Table 5.38. These rules are organized into four sets: QoS composition rules for *throughput* (Table G.1), QoS composition rules for endToEndDelay (Table G.2), QoS decomposition rules for throughput (Table G.3), and QoS decomposition rules for endToEndDelay (Table G.4).

Table G.1 QoS Composition Rules for *throughput* for the Banking Domain Example

QoS Composition Rules for throughput for the Banking Domain Example
System_throughput = [CriticalUseCaseModelInstance]_throughput [CriticalUseCaseModelInstance]_throughput = min ({CriticalUseCase}_throughput) 1/DepositMoneyCase1_1_throughput = 1/CashierTerminal.deposit_throughput + 1/DeluxeTransactionServer.deposit_throughput + 1/AccountDatabase.getAccount_throughput + 1/AccountDatabase.saveAccount_throughput 1/DepositMoneyCase1_2_throughput = 1/CashierTerminal.deposit_throughput + 1/min(DeluxeTransactionServer.deposit_throughput, AccountDatabase.getAccount_throughput, AccountDatabase.saveAccount_throughput) 1/DepositMoneyCase2_throughput = 1/CashierTerminal.deposit_throughput + 1/EconomicServer.deposit_throughput 1/WithdrawMoneyCase1_1_throughput = 1/CashierTerminal.withdraw_throughput + 1/DeluxeTransactionServer.withdraw_throughput + 1/AccountDatabase.getAccount_throughput + 1/AccountDatabase.saveAccount_throughput 1/WithdrawMoneyCase1_2_throughput = 1/CashierTerminalQoS.withdraw.throughput + 1/min(DeluxeTransactionServer.withdraw_throughput, AccountDatabase.getAccount_throughput, AccountDatabase.saveAccount_throughput) 1/WithdrawMoneyCase2_throughput = 1/CashierTerminal.withdraw_throughput + 1/EconomicServer.withdraw_throughput 1/TransferMoneyCase1_1_throughput = 1/CashierTerminal.transfer_throughput + 1/DeluxeTransactionServer.transfer_throughput + 1/AccountDatabase.getAccount_throughput + 1/AccountDatabase.saveAccount_throughput 1/TransferMoneyCase1_2_throughput = 1/CashierTerminal.transfer_throughput + 1/min(DeluxeTransactionServer.transfer_throughput, AccountDatabase.getAccount_throughput, AccountDatabase.saveAccount_throughput) 1/TransferMoneyCase2_throughput = 1/CashierTerminal.transfer_throughput + 1/EconomicServer.transfer_throughput

Table G.2 QoS Composition Rules for *endToEndDelay* for the Banking Domain Example

QoS Composition Rules for endToEndDelay for the Banking Domain Example
SystemQoS.endToEndDelay = [CriticalUseCaseModelInstance]_endToEndDelay [CriticalUseCaseModelInstance]_endToEndDelay = max ({CriticalUseCase}_endToEndDelay) DepositMoneyCase1_1_endToEndDelay = sum(CashierTerminal.deposit_endToEndDelay, DeluxeTransactionServer.deposit_endToEndDelay, AccountDatabase.getAccount_endToEndDelay, AccountDatabase.saveAccount_endToEndDelay) DepositMoneyCase1_2_endToEndDelay = sum(CashierTerminal.deposit_endToEndDelay, DeluxeTransactionServer.deposit_endToEndDelay, AccountDatabase.getAccount_endToEndDelay, AccountDatabase.saveAccount_endToEndDelay) DepositMoneyCase2_endToEndDelay = sum(CashierTerminal.deposit_endToEndDelay, EconomicTransactionServer.deposit_endToEndDelay) WithdrawMoneyCase1_1_endToEndDelay = sum(CashierTerminal.withdraw_endToEndDelay, DeluxeTransactionServer.withdraw_endToEndDelay, AccountDatabase.getAccount_endToEndDelay, AccountDatabase.saveAccount_endToEndDelay) WithdrawMoneyCase1_2_endToEndDelay = sum(CashierTerminal.withdraw_endToEndDelay, DeluxeTransactionServer.withdraw_endToEndDelay, AccountDatabase.getAccount_endToEndDelay, AccountDatabase.saveAccount_endToEndDelay) WithdrawMoneyCase2_endToEndDelay = sum(CashierTerminal.withdraw_endToEndDelay, EconomicTransactionServer.withdraw_endToEndDelay) TransferMoneyCase1_1_endToEndDelay = sum(CashierTerminal.transfer_endToEndDelay, DeluxeTransactionServer.transfer_endToEndDelay, AccountDatabase.getAccount_endToEndDelay, AccountDatabase.saveAccount_endToEndDelay) TransferMoneyCase1_2_endToEndDelay = sum(CashierTerminal.transfer_endToEndDelay, DeluxeTransactionServer.transfer_endToEndDelay, AccountDatabase.getAccount_endToEndDelay, AccountDatabase.saveAccount_endToEndDelay) TransferMoneyCase2_endToEndDelay = sum(CashierTerminal.transfer_endToEndDelay, EconomicTransactionServer.transfer_endToEndDelay)

Table G.3 QoS Decomposition Rules for *throughput*
for the Banking Domain Example

QoS Decomposition Rules for Throughput for Bank
[CriticalUseCaseModelInstance]_throughput > System_throughput
{CriticalUseCase}_throughput > System_throughput
<DepositMoneyCase1_1>_throughput > System_throughput
CashierTerminal.deposit_throughput > System_throughput
DeluxeTransactionServer.deposit_throughput > System_throughput
AccountDatabase.getAccount_throughput > System_throughput
AccountDatabase.saveAccount_throughput > System_throughput
<DepositMoneyCase1_2>_throughput > System_throughput
CashierTerminal.deposit_throughput > System_throughput
DeluxeTransactionServer.deposit_throughput > System_throughput
AccountDatabase.getAccount_throughput > System_throughput
AccountDatabase.saveAccount_throughput > System_throughput
<DepositMoneyCase2>_throughput > System_throughput
CashierTerminal.deposit_throughput > System_throughput
EconomicTransactionServer.deposit_throughput > System_throughput
<WithdrawMoneyCase1_1>_throughput > System_throughput
CashierTerminal.withdraw_throughput > System_throughput
DeluxeTransactionServer.withdraw_throughput > System_throughput
AccountDatabase.getAccount_throughput > System_throughput
AccountDatabase.saveAccount_throughput > System_throughput
<WithdrawMoneyCase1_2>_throughput > System_throughput
CashierTerminal.withdraw_throughput > System_throughput
DeluxeTransactionServer.withdraw_throughput > System_throughput
AccountDatabase.getAccount_throughput > System_throughput
AccountDatabase.saveAccount_throughput > System_throughput
<WithdrawMoneyCase2>_throughput > System_throughput
CashierTerminal.withdraw_throughput > System_throughput
EconomicTransactionServer.withdraw_throughput > System_throughput
<TransferMoneyCase1_1>_throughput > System_throughput
CashierTerminal.transfer_throughput > System_throughput
DeluxeTransactionServer.transfer_throughput > System_throughput
AccountDatabase.getAccount_throughput > System_throughput
AccountDatabase.saveAccount_throughtput > System_throughput
<TransferMoneyCase1_2>_throughput > System_throughput
CashierTerminal.transfer_throughput > System_throughput
DeluxeTransactionServer.transfer_throughput > System_throughput
AccountDatabase.getAccount_throughput > System_throughput
AccountDatabase.saveAccount_throughtput > System_throughput
<TransferMoneyCase2>_throughput > System_throughput
CashierTerminal.transfer_throughput > System_throughput
EconomicTransactionServer.transfer_throughput > System_throughput

Table G.4 QoS Decomposition Rules for *endToEndDelay*
for the Banking Domain Example

[CriticalUseCaseModelInstance]_endToEndDelay < System_endToEndDelay
{CriticalUseCase}_endToEndDelay < System_endToEndDelay

<DeposistMoneyCase1_1>_endToEndDelay < System_endToEndDelay
CashierTerminal.deposit_endToEndDelay < System_endToEndDelay
DeluxeTransactionServer.deposit_endToEndDelay < System_endToEndDelay
AccountDatabase.getAccount_endToEndDelay < System_endToEndDelay
AccountDatabase.saveAccount_endToEndDelay < System_endToEndDelay
<DeposistMoneyCase1_2>_endToEndDelay < System_endToEndDelay
CashierTerminal.deposit_endToEndDelay < System_endToEndDelay
DeluxeTransactionServer.deposit_endToEndDelay < System_endToEndDelay
AccountDatabase.getAccount_endToEndDelay < System_endToEndDelay
AccountDatabase.saveAccount_endToEndDelay < System_endToEndDelay
<DeposistMoneyCase2>_endToEndDelay < System_endToEndDelay
CashierTerminal.deposit_endToEndDelay < System_endToEndDelay
EconomicTransactionServer.deposit_endToEndDelay < System_endToEndDelay
<WithdrawMoneyCase1_1>_endToEndDelay < System_endToEndDelay
CashierTerminal.withdraw_endToEndDelay < System_endToEndDelay
DeluxeTransactionServer.withdraw_endToEndDelay < System_endToEndDelay
AccountDatabase.getAccount_endToEndDelay < System_endToEndDelay
AccountDatabase.saveAccount_endToEndDelay < System_endToEndDelay
<WithdrawMoneyCase1_2>_endToEndDelay < System_endToEndDelay
CashierTerminal.withdraw_endToEndDelay < System_endToEndDelay
DeluxeTransactionServer.withdraw_endToEndDelay < System_endToEndDelay
AccountDatabase.getAccount_endToEndDelay < System_endToEndDelay
AccountDatabase.saveAccount_endToEndDelay < System_endToEndDelay
<WithdrawMoneyCase2>_endToEndDelay < System_endToEndDelay
CashierTerminal.withdraw_endToEndDelay < System_endToEndDelay
EconomicTransactionServer.withdraw_endToEndDelay < System_endToEndDelay
<TransferMoneyCase1_1>_endToEndDelay < System_endToEndDelay
CashierTerminal.transfer_endToEndDelay < System_endToEndDelay
DeluxeTransactionServer.transfer_endToEndDelay < System_endToEndDelay
AccountDatabase.getAccount_endToEndDelay < System_endToEndDelay
AccountDatabase.saveAccount_endToEndDelay < System_endToEndDelay
<TransferMoneyCase1_2>_endToEndDelay < System_endToEndDelay
CashierTerminal.transfer_endToEndDelay < System_endToEndDelay
DeluxeTransactionServer.transfer_endToEndDelay < System_endToEndDelay
AccountDatabase.getAccount_endToEndDelay < System_endToEndDelay
AccountDatabase.saveAccount_endToEndDelay < System_endToEndDelay
<TransferMoneyCase2>_endToEndDelay < System_endToEndDelay
CashierTerminal.transfer_endToEndDelay < System_endToEndDelay
EconomicTransactionServer.transfer_endToEndDelay < System_endToEndDelay

APPENDIX H: QoS Composition and Decomposition Model for the Banking Domain Example

This appendix consists of the QoS Composition and Decomposition Model (QCDM) for the banking domain example. The model consists of the QoS composition and decomposition rules for the three cases of the Critical Use Case Model in disjunctive normal form in the banking domain example. Table H.1 and Table H.2 illustrate the rules for CriticalUseCase1. Table H.3 and H.4 illustrate the rules for CriticalUseCase2. Table H.5 and Table H.6 illustrate the rules for CriticalUseCase3.

Table H.1 QCDM for *CriticalUseCase1*

QoS Composition and Decomposition Model for CriticalUseCase1
1. QoS Composition Rules for throughput System_throughput = CriticaluseCase1_throughput CriticalUseCase1_throughput = min (DepositMoneyCase1_1_throughput, WithdrawMoneyCase1_1_throughput, TransferMoneyCase1_1_throughput) 1/DepositMoneyCase1_1_throughput = 1/CashierTerminal.deposit_throughput + 1/DeluxeTransactionServer.deposit_throughput + 1/AccountDatabase.getAccount_throughput + 1/AccountDatabase.saveAccount_throughput 1/WithdrawMoneyCase1_1_throughput = 1/CashierTerminal.withdraw_throughput + 1/DeluxeTransactionServer.withdraw_throughput + 1/AccountDatabase.getAccount_throughput + 1/AccountDatabase.saveAccount_throughput 1/TransferMoneyCase1_1_throughput = 1/CashierTerminal.transfer_throughput + 1/DeluxeTransactionServer.transfer_throughput + 1/AccountDatabase.getAccount_throughput + 1/AccountDatabase.saveAccount_throughput
2. QoS Composition Rules for endToEndDelay SystemQoS.endToEndDelay = CriticalUseCase1_endToEndDelay CriticalUseCase1_endToEndDelay = max (DepositMoneyCase1_1_endToEndDelay, WithdrawMoneyCase1_1_endToEndDelay, TransferMoneyCase1_1_endToEndDelay) DepositMoneyCase1_1_endToEndDelay = sum(CashierTerminal.deposit_endToEndDelay, DeluxeTransactionServer.deposit_endToEndDelay, AccountDatabase.getAccount_endToEndDelay, AccountDatabase.saveAccount_endToEndDelay) WithdrawMoneyCase1_1_endToEndDelay = sum(CashierTerminal.withdraw_endToEndDelay, DeluxeTransactionServer.withdraw_endToEndDelay, AccountDatabase.getAccount_endToEndDelay, AccountDatabase.saveAccount_endToEndDelay) TransferMoneyCase1_1_endToEndDelay = sum(CashierTerminal.transfer_endToEndDelay, DeluxeTransactionServer.transfer_endToEndDelay, AccountDatabase.getAccount_endToEndDelay, AccountDatabase.saveAccount_endToEndDelay)
(Continued in Table H.2)

Table H.2 QCDM for *CriticalUseCase1* (Continued from Table H.1)

QoS Composition and Decomposition Model for CriticalUseCase1

(Continued from Table H.1)

3. QoS Decomposition Rules for throughput
CriticalUseCase1_throughput > System_throughput
DepositMoneyCase1_1_throughput > System_throughput
WithdrawMoneyCase1_1_throughput > System_throughput
TransferMoenyCase1_1_throughput > System_throughput

<DepositMoneyCase1_1>_throughput > System_throughput
CashierTerminal.deposit_throughput > System_throughput
DeluxeTransactionServer.deposit_throughput > System_throughput
AccountDatabase.getAccount_throughput > System_throughput
AccountDatabase.saveAccount_throughput > System_throughput
<WithdrawMoneyCase1_1>_throughput > System_throughput
CashierTerminal.withdraw_throughput > System_throughput
DeluxeTransactionServer.withdraw_throughput > System_throughput
AccountDatabase.getAccount_throughput > System_throughput
AccountDatabase.saveAccount_throughput > System_throughput
<TransferMoneyCase1_1>_throughput > System_throughput
CashierTerminal.transfer_throughput > System_throughput
DeluxeTransactionServer.transfer_throughput > System_throughput
AccountDatabase.getAccount_throughput > System_throughput
AccountDatabase.saveAccount_throughtput > System_throughput

4. QoS Decomposition Rules for endToEndDelay
CriticalUseCase1_endToEndDelay < System_endToEndDelay
DepositMoneyCase1_1_endToEndDelay < System_endToEndDelay
WithdrawMoneyCase1_1_endToEndDelay < System_endToEndDelay
TransferMoenyCase1_1_endToEndDelay < System_endToEndDelay

<DeposistMoneyCase1_1>_endToEndDelay < System_endToEndDelay
CashierTerminal.deposit_endToEndDelay < System_endToEndDelay
DeluxeTransactionServer.deposit_endToEndDelay < System_endToEndDelay
AccountDatabase.getAccount_endToEndDelay < System_endToEndDelay
AccountDatabase.saveAccount_endToEndDelay < System_endToEndDelay
<WithdrawMoneyCase1_1>_endToEndDelay < System_endToEndDelay
CashierTerminal.withdraw_endToEndDelay < System_endToEndDelay
DeluxeTransactionServer.withdraw_endToEndDelay < System_endToEndDelay
AccountDatabase.getAccount_endToEndDelay < System_endToEndDelay
AccountDatabase.saveAccount_endToEndDelay < System_endToEndDelay
<TransferMoneyCase1_1>_endToEndDelay < System_endToEndDelay
CashierTerminal.transfer_endToEndDelay < System_endToEndDelay
DeluxeTransactionServer.transfer_endToEndDelay < System_endToEndDelay
AccountDatabase.getAccount_endToEndDelay < System_endToEndDelay
AccountDatabase.saveAccount_endToEndDelay < System_endToEndDelay

Table H.3 QCDM for *CriticalUseCase2*

QoS Composition and Decomposition Model for CriticalUseCase2
1. QoS Composition Rules for throughput System_throughput = CriticalUseCase2_throughput CriticalUseCase2_throughput = min (DepositMoneyCase1_2_throughput, WithdrawMoneyCase1_2_throughput, TransferMoneyCase1_2_throughput) 1/DepositMoneyCase1_2_throughput = 1/CashierTerminal.deposit_throughput + 1/min(DeluxeTransactionServer.deposit_throughput, AccountDatabase.getAccount_throughput, AccountDatabase.saveAccount_throughput) 1/WithdrawMoneyCase1_2_throughput = 1/CashierTerminalQoS.withdraw.throughput + 1/min(DeluxeTransactionServer.withdraw_throughput, AccountDatabase.getAccount_throughput, AccountDatabase.saveAccount_throughput) 1/TransferMoneyCase1_2_throughput = 1/CashierTerminal.transfer_throughput + 1/min(DeluxeTransactionServer.transfer_throughput, AccountDatabase.getAccount_throughput, AccountDatabase.saveAccount_throughput)
2. QoS Composition Rules for endToEndDelay SystemQoS.endToEndDelay = CriticalUseCase2_endToEndDelay CriticalUseCase2_endToEndDelay = max (DepositMoneyCase1_2_endToEndDelay, WithdrawMoneyCase1_2_endToEndDelay, TransferMoneyCase1_2_endToEndDelay) DepositMoneyCase1_2_endToEndDelay = sum(CashierTerminal.deposit_endToEndDelay, DeluxeTransactionServer.deposit_endToEndDelay, AccountDatabase.getAccount_endToEndDelay, AccountDatabase.saveAccount_endToEndDelay) WithdrawMoneyCase1_2_endToEndDelay = sum(CashierTerminal.withdraw_endToEndDelay, DeluxeTransactionServer.withdraw_endToEndDelay, AccountDatabase.getAccount_endToEndDelay, AccountDatabase.saveAccount_endToEndDelay) TransferMoneyCase1_2_endToEndDelay = sum(CashierTerminal.transfer_endToEndDelay, DeluxeTransactionServer.transfer_endToEndDelay, AccountDatabase.getAccount_endToEndDelay, AccountDatabase.saveAccount_endToEndDelay)
(Continued in Table H.4)

Table H.4 QCDM for *CriticalUseCase2* (Continued from Table H.3)

QoS Composition and Decomposition Model for CriticalUseCase2

(Continued from Table H.3)

3. QoS Decomposition Rules for throughput
CriticalUseCase2_throughput > System_throughput
DepositMoneyCase1_2_throughput > System_throughput
WithdrawMoneyCase1_2_throughput > System_throughput
TransferMoenyCase1_2_throughput > System_throughput

<DepositMoneyCase1_2>_throughput > System_throughput
CashierTerminal.deposit_throughput > System_throughput
DeluxeTransactionServer.deposit_throughput > System_throughput
AccountDatabase.getAccount_throughput > System_throughput
AccountDatabase.saveAccount_throughput > System_throughput
<WithdrawMoneyCase1_2>_throughput > System_throughput
CashierTerminal.withdraw_throughput > System_throughput
DeluxeTransactionServer.withdraw_throughput > System_throughput
AccountDatabase.getAccount_throughput > System_throughput
AccountDatabase.saveAccount_throughput > System_throughput
<TransferMoneyCase1_2>_throughput > System_throughput
CashierTerminal.transfer_throughput > System_throughput
DeluxeTransactionServer.transfer_throughput > System_throughput
AccountDatabase.getAccount_throughput > System_throughput
AccountDatabase.saveAccount_throughtput > System_throughput

4. QoS Decomposition Rules for endToEndDelay
CriticalUseCase2_endToEndDelay < System_endToEndDelay
DepositMoneyCase1_2_endToEndDelay < System_endToEndDelay
WithdrawMoneyCase1_2_endToEndDelay < System_endToEndDelay
TransferMoenyCase1_2_endToEndDelay < System_endToEndDelay

<DeposistMoneyCase1_2>_endToEndDelay < System_endToEndDelay
CashierTerminal.deposit_endToEndDelay < System_endToEndDelay
DeluxeTransactionServer.deposit_endToEndDelay < System_endToEndDelay
AccountDatabase.getAccount_endToEndDelay < System_endToEndDelay
AccountDatabase.saveAccount_endToEndDelay < System_endToEndDelay
<WithdrawMoneyCase1_2>_endToEndDelay < System_endToEndDelay
CashierTerminal.withdraw_endToEndDelay < System_endToEndDelay
DeluxeTransactionServer.withdraw_endToEndDelay < System_endToEndDelay
AccountDatabase.getAccount_endToEndDelay < System_endToEndDelay
AccountDatabase.saveAccount_endToEndDelay < System_endToEndDelay
<TransferMoneyCase1_2>_endToEndDelay < System_endToEndDelay
CashierTerminal.transfer_endToEndDelay < System_endToEndDelay
DeluxeTransactionServer.transfer_endToEndDelay < System_endToEndDelay
AccountDatabase.getAccount_endToEndDelay < System_endToEndDelay
AccountDatabase.saveAccount_endToEndDelay < System_endToEndDelay

Table H.5 QCDM for *CriticalUseCase3*

QoS Composition and Decomposition Model for CriticalUseCase3

1. QoS Composition Rules for throughput
System_throughput = CriticalUseCase3_throughput
CriticalUseCase3_throughput = min (DepositMoneyCase2_throughput,
WithdrawMoneyCase2_throughput, TransferMoneyCase2_throughput)
1/DepositMoneyCase2_throughput = 1/CashierTerminal.deposit_throughput +
1/EconomicServer.deposit_throughput
1/WithdrawMoneyCase2_throughput = 1/CashierTerminal.withdraw_throughput +
1/EconomicServer.withdraw_throughput
1/TransferMoneyCase2_throughput = 1/CashierTerminal.transfer_throughput +
1/EconomicServer.transfer_throughput

2. QoS Composition Rules for endToEndDelay
SystemQoS.endToEndDelay = CriticalUseCase3_endToEndDelay
CriticalUseCase3_endToEndDelay = max (DepositMoneyCase2_endToEndDelay,
WithdrawMoneyCase2_endToEndDelay, TransferMoneyCase2_endToEndDelay)
DepositMoneyCase2_endToEndDelay = sum(CashierTerminal.deposit_endToEndDelay,
EconomicTransactionServer.deposit_endToEndDelay)
WithdrawMoneyCase2_endToEndDelay = sum(CashierTerminal.withdraw_endToEndDelay,
EconomicTransactionServer.withdraw_endToEndDelay)
TransferMoneyCase2_endToEndDelay = sum(CashierTerminal.transfer_endToEndDelay,
EconomicTransactionServer.transfer_endToEndDelay)

3. QoS Decomposition Rules for throughput
CriticalUseCase3_throughput > System_throughput
DepositMoneyCase2_throughput > System_throughput
WithdrawMoneyCase2_throughput > System_throughput
TransferMoenyCase2_throughput > System_throughput

<DepositMoneyCase2>_throughput > System_throughput
CashierTerminal.deposit_throughput > System_throughput
EconomicTransactionServer.deposit_throughput > System_throughput
<WithdrawMoneyCase2>_throughput > System_throughput
CashierTerminal.withdraw_throughput > System_throughput
EconomicTransactionServer.withdraw_throughput > System_throughput
<TransferMoneyCase2>_throughput > System_throughput
CashierTerminal.transfer_throughput > System_throughput
EconomicTransactionServer.transfer_throughput > System_throughput

(Continued in Table H.6)

Table H.6 QCDM for *CriticalUseCase3* (Continued from Table H.5)

QoS Composition and Decomposition Model for CriticalUseCase3

(Continued from Table H.5)

4. QoS Decomposition Rules for endToEndDelay
CriticalUseCase3_endToEndDelay < System_endToEndDelay
DepositMoneyCase2_endToEndDelay < System_endToEndDelay
WithdrawMoneyCase2_endToEndDelay < System_endToEndDelay
TransferMoenyCase2_endToEndDelay < System_endToEndDelay

<DeposistMoneyCase2>_endToEndDelay < System_endToEndDelay
    CashierTerminal.deposit_endToEndDelay < System_endToEndDelay
    EconomicTransactionServer.deposit_endToEndDelay < System_endToEndDelay
<WithdrawMoneyCase2>_endToEndDelay < System_endToEndDelay
    CashierTerminal.withdraw_endToEndDelay < System_endToEndDelay
    EconomicTransactionServer.withdraw_endToEndDelay < System_endToEndDelay
<TransferMoneyCase2>_endToEndDelay < System_endToEndDelay
    CashierTerminal.transfer_endToEndDelay < System_endToEndDelay
EconomicTransactionServer.transfer_endToEndDelay < System_endToEndDelay

APPENDIX I: UGDM in XML Format for the Banking Domain Example

This appendix consists of various models of the UGDM in the XML format for the banking domain example. The models documented in this appendix include Architecture Model in Disjunctive Normal Form (AMDNF) at the component level (Table I.1 and Table I.2), Architecture Model in Disjunctive Normal Form (AMDNF) at the function/interface level (Table I.3, Table I.4, Table I.5 and Table I.6), Abstract Component Interaction Model (Table I.7), Architecture Model in Disjunctive Normal Form and Critical Use Case Model Mapping (Function/Interface Level) (Table I.8) and the Mapping of AMDNF from Component Level to Function/Interface Level (Table I.9).

Table I.1 AMDNF at Component Level in the XML Format


```xml
<?xml version="1.0" encoding='utf-8'?>

<!-- architecture at component level for the banking domain example -->

<architecture_component>
   <system_name> Bank </system_name>
   <case>
      <case_name> BankCase1 </case_name>
      <component> ATM </component>
      <component> CashierTerminal </component>
      <component> CustomerValidationServer </component>
      <component> CashierValidationServer </component>
      <component> TransactionServerManager </component>
      <component> EconomicTransactionServer </component>
   </case>
   <case>
      <case_name> BankCase2 </case_name>
      <component> ATM </component>
      <component> CashierTerminal </component>
      <component> CustomerValidationServer </component>
      <component> CashierValidationServer </component>
      <component> TransactionServerManager </component>
      <component> DeluxeTransactionServer </component>
      <component> AccountDatabase </component>
   </case>
```


(Continued in Table I.2)

Table I.2 AMDNF at Component Level in the XML Format
(Continued from Table I.1)

(Continued from Table I.1)

```
  <case>
    <case_name> BankCase3 </case_name>
    <component> CashierTerminal </component>
    <component> CashierValidationServer </component>
    <component> TransactionServerManager </component>
    <component> EconomicTransactionServer </component>
  </case>
  <case>
    <case_name> BankCase4 </case_name>
    <component> CashierTerminal </component>
    <component> CashierValidationServer </component>
    <component> TransactionServerManager </component>
    <component> DeluxeTransactionServer </component>
    <component> AccountDatabase </component>
  </case>
</architecture_component>
```

Table I.3 AMDNF at Function/Interface Level in the XML Format

```
<?xml version="1.0" encoding='utf-8'?>

<!-- architecture at interface level for the banking domain example -->

<architecture_interface>
  <system_name> Bank </system_name>
  <case>
    <case_name> BankCase1 </case_name>
    <component>
      <componentname> ATM </componentname>
      <componentsubcase> ATMCase1 </componentsubcase>
    </component>
    <component>
      <componentname> CashierTerminal </componentname>
      <componentsubcase> CashierTerminalCase1 </componentsubcase>
    </component>
    <component>
      <componentname> CustomerValidationServer </componentname>
      <componentsubcase> CustomerValidationServerCase1 </componentsubcase>
    </component>
    <component>
      <componentname> CashierValidationServer </componentname>
      <componentsubcase> CashierValidationServerCase1 </componentsubcase>
    </component>
```

(Continued in Table I.4)

Table I.4 AMDNF at Function/Interface Level in the XML Format
(Continued from Table I.3)

(Continued from Table I.3)

```
    <component>
      <componentname> TransactionServerManager </componentname>
      <componentsubcase> TransactionServerManagerCase1 </componentsubcase>
    </component>
    <component>
      <componentname> EconomicTransactionServer </componentname>
      <componentsubcase> EconomicTransactionServerCase1 </componentsubcase>
    </component>
  </case>
  <case>
    <case_name> BankCase2_1 </case_name>
    <component>
      <componentname> ATM </componentname>
      <componentsubcase> ATMCase1 </componentsubcase>
    </component>
    <component>
      <componentname> CashierTerminal </componentname>
      <componentsubcase> CashierTerminalCase1 </componentsubcase>
    </component>
    <component>
      <componentname> CustomerValidationServer </componentname>
      <componentsubcase> CustomerValidationServerCase1 </componentsubcase>
    </component>
    <component>
      <componentname> CashierValidationServer </componentname>
      <componentsubcase> CashierValidationServerCase1 </componentsubcase>
    </component>
    <component>
      <componentname> TransactionServerManager </componentname>
      <componentsubcase> TransactionServerManagerCase1 </componentsubcase>
    </component>
    <component>
      <componentname> DeluxeTransactionServer </componentname>
      <componentsubcase> DeluxeTransactionServerCase1 </componentsubcase>
    </component>
    <component>
      <componentname> AccountDatabase </componentname>
      <componentsubcase> AccountDatabaseCase1 </componentsubcase>
    </component>
  </case>
  <case>
    <case_name> BankCase2_2 </case_name>
    <component>
      <componentname> ATM </componentname>
      <componentsubcase> ATMCase1 </componentsubcase>
    </component>
```

(Continued in Table I.5)

Table I.5 AMDNF at Function/Interface Level in the XML Format
(Continued from Table I.4)

(Continued from Table I.4)

```
   <component>
      <componentname> CashierTerminal </componentname>
      <componentsubcase> CashierTerminalCase1 </componentsubcase>
   </component>
   <component>
      <componentname> CustomerValidationServer </componentname>
      <componentsubcase> CustomerValidationServerCase1 </componentsubcase>
   </component>
   <component>
      <componentname> CashierValidationServer </componentname>
      <componentsubcase> CashierValidationServerCase1 </componentsubcase>
   </component>
   <component>
      <componentname> TransactionServerManager </componentname>
      <componentsubcase> TransactionServerManagerCase1 </componentsubcase>
   </component>
   <component>
      <componentname> DeluxeTransactionServer </componentname>
      <componentsubcase> DeluxeTransactionServerCase2 </componentsubcase>
   </component>
   <component>
      <componentname> AccountDatabase </componentname>
      <componentsubcase> AccountDatabaseCase2 </componentsubcase>
   </component>
</case>
<case>
   <case_name> BankCase3 </case_name>
   <component>
      <componentname> CashierTerminal </componentname>
      <componentsubcase> CashierTerminalCase1 </componentsubcase>
   </component>
   <component>
      <componentname> CashierValidationServer </componentname>
      <componentsubcase> CashierValidationServerCase1 </componentsubcase>
   </component>
   <component>
      <componentname> TransactionServerManager </componentname>
      <componentsubcase> TransactionServerManagerCase1 </componentsubcase>
   </component>
   <component>
      <componentname> EconomicTransactionServer </componentname>
      <componentsubcase> EconomicTransactionServerCase1 </componentsubcase>
   </component>
</case>
```

(Continued in Table I.6)

Table I.6 AMDNF at Function/Interface Level in the XML Format
(Continued from Table I.5)

(Continued from Table I.5)


```xml
  <case>
    <case_name> BankCase4_1 </case_name>
    <component>
      <componentname> CashierTerminal </componentname>
      <componentsubcase> CashierTerminalCase1 </componentsubcase>
    </component>
    <component>
      <componentname> CashierValidationServer </componentname>
      <componentsubcase> CashierValidationServerCase1 </componentsubcase>
    </component>
    <component>
      <componentname> TransactionServerManager </componentname>
      <componentsubcase> TransactionServerManagerCase1 </componentsubcase>
    </component>
    <component>
      <componentname> DeluxeTransactionServer </componentname>
      <componentsubcase> DeluxeTransactionServerCase1 </componentsubcase>
    </component>
    <component>
      <componentname> AccountDatabase </componentname>
      <componentsubcase> AccountDatabaseCase1 </componentsubcase>
    </component>
  </case>
  <case>
    <case_name> BankCase4_2 </case_name>
    <component>
      <componentname> CashierTerminal </componentname>
      <componentsubcase> CashierTerminalCase1 </componentsubcase>
    </component>
    <component>
      <componentname> CashierValidationServer </componentname>
      <componentsubcase> CashierValidationServerCase1 </componentsubcase>
    </component>
    <component>
      <componentname> TransactionServerManager </componentname>
      <componentsubcase> TransactionServerManagerCase1 </componentsubcase>
    </component>
    <component>
      <componentname> DeluxeTransactionServer </componentname>
      <componentsubcase> DeluxeTransactionServerCase2 </componentsubcase>
    </component>
    <component>
      <componentname> AccountDatabase </componentname>
      <componentsubcase> AccountDatabaseCase2 </componentsubcase>
    </component>
  </case>
</architecture_interface>
```

Table I.7 Abstract Component Interaction Model in the XML Format


```xml
<?xml version="1.0" encoding='utf-8'?>

<!-- component interaction model for the banking domain example -->

<component_interaction>
  <system_name> Bank </system_name>
  <interaction>
     <initiator> CashierTerminal </initiator>
     <responder> CashierValidationServer </responder>
  </interaction>
  <interaction>
     <initiator> ATM </initiator>
     <responder> CustomerValiationServer </responder>
  </interaction>
  <interaction>
     <initiator> CashierTerminal </initiator>
     <responder> TransactionServerManager </responder>
  </interaction>
  <interaction>
     <initiator> CashierTerminal </initiator>
     <responder> EconomicTransactionServer </responder>
  </interaction>
  <interaction>
     <initiator> CashierTerminal </initiator>
     <responder> DeluxeTransactionServer </responder>
  </interaction>
  <interaction>
     <initiator> ATM  </initiator>
     <responder>TransactionServerManager </responder>
  </interaction>
  <interaction>
     <initiator> ATM </initiator>
     <responder> EconomicTransactionServer </responder>
  </interaction>
  <interaction>
     <initiator> ATM </initiator>
     <responder> DeluxeTransactionServer </responder>
  </interaction>
  <interaction>
     <initiator> DeluxeTransactionServer </initiator>
     <responder> AccountDatabase </responder>
  </interaction>
</component_interaction>
```

Table I.8 Architecture Model and Critical Use Case Model Mapping (Function/Interface Level) in the XML format


```xml
<?xml version="1.0" encoding='utf-8'?>

<!-- mapping from architecture interface level to critical use case model for the banking
domain example -->

<map_architecture_cucm>
  <system_name> Bank </system_name>
  <map>
     <casenamefrom> BankCase1 </casenamefrom>
     <casenameto> CriticalUseCaseModel3 </casenameto>
  </map>
  <map>
     <casenamefrom> BankCase2_1 </casenamefrom>
     <casenameto> CriticalUseCaseModel1 </casenameto>
  </map>
  <map>
     <casenamefrom> BankCase2_2 </casenamefrom>
     <casenameto> CriticalUseCaseModel2 </casenameto>
  </map>
  <map>
     <casenamefrom> BankCase3 </casenamefrom>
     <casenameto> CriticalUseCaseModel3 </casenameto>
  </map>
  <map>
     <casenamefrom> BankCase4_1 </casenamefrom>
     <casenameto> CriticalUseCaseModel1 </casenameto>
  </map>
  <map>
     <casenamefrom> BankCase4_2 </casenamefrom>
     <casenameto> CriticalUseCaseModel2 </casenameto>
  </map>
</map_architecture_cucm>
```

Table I.9 Mapping of AMDNF from Component Level to Function/Interface Level in the XML Format


```xml
<?xml version="1.0" encoding='utf-8'?>

<!-- Mapping from architecture at component level to architecture at interface level for the
banking domain example -->

<map_architecture>
  <system_name> Bank </system_name>
  <map>
    <casenamefrom> BankCase1 </casenamefrom>
    <casenameto> BankCase1 </casenameto>
  </map>
  <map>
    <casenamefrom> BankCase2 </casenamefrom>
    <casenameto> BankCase2_1 </casenameto>
  </map>
  <map>
    <casenamefrom> BankCase3 </casenamefrom>
    <casenameto> BankCase3 </casenameto>
  </map>
  <map>
    <casenamefrom> BankCase4 </casenamefrom>
    <casenameto> BankCase4_1 </casenameto>
  </map>
</map_architecture>
```

APPENDIX J: UGDM Example: Banking Domain Example

This appendix consists of a complete UGDM for the banking domain example developed in Chapter 5.

UGDM Example: Banking Domain Example

1. General Information
1.1 Domain Name: /Finance/Banking
1.2 System Family Name: Bank
1.3 Version: v1.0
1.4 Date: 10/1/2002
1.5 Author: Zhisheng Huang, UniFrame Research Group
1.6 Description: This system family in the banking domain provides basic account transaction service.

2. Problem Space
2.1 Use Case Model
- Commonality and Variation
  Bank: all (ManageCustomers, ManageAccounts, Login-exitAccount, ValidateUsers)
  ManageCustomers: all (OpenAccount, CloseAccount)
  ManageAccounts: all (ManageAccounts_Cashier, ManageAccounts_Customer?)
  ManageAccounts_Cashier: all (WithdrawMoney_Cashier, DepositMoney_Cashier, TransferMoney_Cashier, CheckBalance_Cashier)
  ManageAccounts_Customer: all (WithdrawMoney_Customer, DepositMoney_Customer, TransferMoney_Customer, CheckBalance_Customer)
  ValidateUsers: all (ValidateUsers_Cashier, ValidateUsers_Customer?)
  Login-exitAccount: all (Login-exitAccount_Cashier, Login-exitAccount_Customer?)
- Constraint Expression
  - Default Constraint
    default (ManageAccounts: ManageAccounts_Cashier)
    default (ValidateUsers: ValidateUsers_Cashier)
    default (Login-exitAccount: Login-exitAccount_Cashier)
  - Satisfaction Constraint
    mutual_require (ValidateUsers_Customer, ManageAccounts_Customer, Login-exitAccount_Customer)

2.2 QoS Requirement Model
System.QoS: all (System.QoS.throughput, System.QoS.endToEndDelay)
System.QoS.throughput: CriticalUseCaseModel.QoS.thoughput
SystemQoS.endToEndDelay: CriticalUseCaseModel.QoS.endToEndDelay

2.3 Architecture Model in Hierarchical Form
- Commonality and Variation
  Bank: all (UserSubsystem, UserValidationSubsystem, TransactionSubsystem)
  UserSubsystem: all (ATM?, CashierTerminal)
  UserValidationSubsystem: all (CustomerValidationServer?, CashierValidationServer)
  TransactionSubsystem: all (TransactionServerManager, one-of (EconomicTransactionSubsystem, DeluxeTransactionSubsystem))
  EconomicTransactionSubsystem: EconomicTransactionServer

DeluxeTransactionSubsystem: all (DeluxeTransactionServer, AccountDatabase)

- Constraint Expression
  - Default Constraint
    default (UserSubsystem: CashierTerminal)
    default (UserValidationSubsystem: CashierValidationServer)
    default (TransactionSubsystem: all (TransactionServerManager,
    EconomicTransactionSubsystem)
  - Satisfaction Constraint
    mutual_require (ATM, CustomerValiationServer)

2.4 System-Level Multiplicity Model
multiplicity ((Bank, CashierTerminal): 1..*)
multiplicity ((Bank, ATM) : 0..*)
multiplicity ((Bank, CashierValidationServer) : 1)
multiplicity ((Bank, CustomerValidationServer) : 0..1)
multiplicity ((Bank, TransactionServerManager) : 1)
multiplicity ((Bank, EconomicTransactionServer) : 0..2)
multiplicity ((Bank, DeluxeTransactionServer) : 0..2)
multiplicity ((Bank, AccountDatabase) : 0..2)

3. Solution Space and Configuration Knowledge
3.1 Architecture Related Models
3.1.1 Architecture Model in Disjunctive Normal Form (Abstract Component Level)

- Disjunctive Normal Form
  Bank: one-of (BankCase1, BankCase2, BankCase3, BankCase4)

  BankCase1: all (ATM, CashierTerminal, CustomerValidationServer, CashierValidationServer,
  TransactionServerManager, EconomicTransactionServer)
  BankCase2: all (ATM, CashierTerminal, CustomerValidationServer, CashierValidationServer,
  TransactionServerManager, DeluxeTransactionServer, AccountDatabase)
  BankCase3: all (CashierTerminal, CashierValidationServer, TransactionServerManager,
  EconomicTransactionServer)
  BankCase4: all (CashierTerminal, CashierValidationServer, TransactionServerManager,
  DeluxeTransactionServer, AccountDatabase)
- Constraint Expression
  - Default Constraint
    Default (Bank: BankCase3)

3.1.2 Architecture Model in Disjunctive Normal Form (Function/Interface Level)

- Disjunctive Normal Form
  Bank: one-of (BankCase1, BankCase2, BankCase3, BankCase4)
  BankCase1: one-of (BankCase1_1)
  BankCase2: one-of (BankCase2_1, BankCase2_2)
  BankCase3: one-of (BankCase3_1)
  BankCase4: one-of (BankCase4_1, BankCase4_2)

  BankCase1_1: all (ATMCase1, CashierTerminalCase1, CustomerValidationServerCase1,
  CashierValidationServerCase1, TransactionServerManagerCase1,
  EconomicTransactionServerCase1)
  BankCase2_1: all (ATMCase1, CashierTerminalCase1, CustomerValidationServerCase1,
  CashierValidationServerCase1, TransactionServerManagerCase1,
  DeluxeTransactionServerCase1, AccountDatabaseCase1)
  BankCase2_2: all (ATMCase1, CashierTerminalCase1, CustomerValidationServerCase1,
  CashierValidationServerCase1, TransactionServerManagerCase1,

DeluxeTransactionServerCase2, AccountDatabaseCase2)
BankCase3_1: all (CashierTerminalCase1, CashierValidationServerCase1, TransactionServerManagerCase1, EconomicTransactionServerCase1)
BankCase4_1: all (CashierTerminalCase1, CashierValidationServerCase1, TransactionServerManager Case1, DeluxeTransactionServerCase1, AccountDatabaseCase1)
BankCase4_2: all (CashierTerminalCase1, CashierValidationServerCase1, TransactionServerManagerCase1, DeluxeTransactionServerCase2, AccountDatabaseCase2)

- Constraint Expression
  - Default Constraint
    default (BankCase2: BankCase2_1)
    default (BankCase4: BankCase4_1)

3.1.3 Architecture Model Mapping
map (BankCase1: BankCase1_1)
map (BankCase2: BankCase2_1)
map (BankCase3: BankCase3_1)
map (BankCase4: BankCase4_1)

3.1.4 Abstract Component Interaction Model
interact (CashierTerminal, CashierValidationServer)
interact (ATM, CustomerValiationServer)
interact (CashierTerminal, TransactionServerManager)
interact (CashierTerminal, EconomicTransactionServer)
interact (CashierTerminal, DeluxeTransactionServer)
interact (ATM, TransactionServerManager)
interact (ATM, EconomicTransactionServer)
interact (ATM, DeluxeTransactionServer)
interact (DeluxeTransactionServer, AccountDatabase)

3.1.5 Component-level Multiplicity Model
multiplicity ((CashierValidationServer, CashierTerminal) : 1..*)
multiplicity ((CustomerValiationServer, ATM) : 1..*)
multiplicity ((TransactionServerManager, CashierTerminal) : 1..*)
multiplicity ((EconomicTransactionServer, CashierTerminal) : 1..*)
multiplicity ((DeluxeTransactionServer, CashierTerminal) : 1..*)
multiplicity ((TransactionServerManager, ATM) : 1..*)
multiplicity ((EconomicTransactionServer, ATM) : 1..*)
multiplicity ((DeluxeTransactionServer, ATM) : 1..*)
multiplicity ((DeluxeTransactionServer, AccountDatabase) : 1)

3.2 Design Feature Related Models
3.2.1 Interface Model
Interface: *IAccountDatabase*
1. Syntax
Account getAccount(String accountNumber, int accountType);
Pre: NONE
Post: NONE
Invariant: NONE
Communication Pattern: cp2s or cp2a
Description: This function returns an account object as identified by the parameters. It returns null if the account specified does not exist.
void saveAccount(Account account);
Pre: NONE
Post: NONE

Invariant: NONE
Communication Pattern: cp2s or cp2a
Description: This function updates the account if it already exists; otherwise it adds an entry in the database for this new account.

void removeAccount(Account account, int accountType);
Pre: NONE
Post: NONE
Invariant: NONE
Communication Pattern: cp2s or cp2a
Description: This function removes the specified account if it exists; otherwise it does nothing.

2. Variation
IAccountDatabase: one-of (IAccountDatabaseCase1, IAccountDatabaseCase2)
IAccountDatabaseCase1: {cp2s}
IAccountDatabaseCase2: {cp2a}

3. Default
default (IAccountDatabase: IAccountDatabaseCase1)

Interface: *IAccountManagement*
…
Interface: *ICustomerManagement*
…
Interface: *ITransactionServerManager*
…
Interface: *IValidation*
…

3.2.2 Abstract Component Interface Model

- Disjunctive Normal Form

CashierTerminal: CashierTerminalCase1
ATM: ATMCase1
CashierValidationServer: CashierValidationServerCase1
CustomerValidationServer: CustomerValidationServerCase1
TransactionServerManager: TransactionServerManagerCase1
EconomicTransactionServer: EconomicTransactionServerCase1
DeluxeTransactionServer: one-of (DeluxeTransaxtionServerCase1, DeluxeTransactionServerCase2)
AccountDatabase: one-of (AccountDatabaseCase1, AccountDatabaseCase2)

interface (CashierTerminalCase1: provided_interface (ICustomerManagementCase1, IAccountManagementCase1), required_interface (ICustomerManagementCase1, IAccountManagementCase1, ITransactionServerManagerCase1, IValidationCase1))
interface (ATMCase1: provided_interface (IAccountManagementCase1), required_interface (IAccountManagementCase1, ITransactionServerManagerCase1, IValidationCase1))
interface (CashierValidationServerCase1: provided_interface (IValidationCase1), required_interface (NONE))
interface (CustomerValidationServerCase1: provided_interface (IValidationCase1), required_interface (NONE))
interface (TransactionServerManagerCase1: provided_interface (ITransactionServerManagerCase1), required_interface (NONE))
interface (EconomicTransactionServerCase1: provided_interface (IAccountManagementCase1, ICustomerManagementCase1), required_inteface (NONE))
interface (DeluxeTransaxtionServerCase1: provided_interface (IAccountManagementCase1, ICustomerManagementCase1), required_interface (IAccountDatabaseCase1))

interface (DeluxeTransactionServerCase2: provided_interface (IAccountManagementCase1, ICustomerManagementCase1), required_interface (IAccountDatabaseCase2))
interface (AccountDatabaseCase1: provided_interface (IAccountDatabaseCase1), required_interface (NONE))
interface (AccountDatabaseCase2: provided_interface (IAccountDatabaseCase2), required_interface (NONE))

- Constraint Expression
  - Default Constraint
    default (DeluxeTransactionServer : DeluxeTransactionServerCase1)
    default (AccountDatabase : AccountDatabaseCase1)
  - Satisfaction Constraint
    mutual_require (DeluxeTransactionServerCase1, AccountDatabaseCase1)
    mutual_require (DeluxeTransactionServerCase2, AccountDatabaseCase2)

3.2.3 Abstract Component Model

Abstract Component**:** *AccountDatabaseCase1*

1. Component Name: *AccountDatabase*
2. Component Subcase: *AccountDatabaseCase1*
3. Domain Name: Banking
4. System Name: Bank
5. Informal Description: Provide an account database service.
6. Computational Attributes:
   6.1 Inherent Attributes:
      6.1.1 id: N/A
      6.1.2 Version: version 1.0
      6.1.3 Author: N/A
      6.1.4 Date: N/A
      6.1.5 Validity: N/A
      6.1.6 Atomicity: Yes
      6.1.7 Registration: N/A
      6.1.8 Model: N/A
   6.2 Functional Attributes:
      6.2.1 Function description: Serve as an account database.
      6.2.2 Algorithm: N/A
      6.2.3 Complexity: N/A
      6.2.4 Syntactic Contract
         6.2.4.1 Provided Interface: IAccountDatabaseCase1
         6.2.4.2 Required Interface: NONE
      6.2.5 Technology: N/A
      6.2.6 Expected Resources: N/A
      6.2.7 Design Patterns: NONE
      6.2.8 Known Usage: NONE
      6.2.9 Alias: NONE
7. Cooperation Attributes
   7.1 Preprocessing Collaborators: DeluxeTransactionServerCase1
   7.2 Postprocessing Collaborators: NONE
8. Auxiliary Attributes:
   8.1 Mobility: No
   8.2 Security: *L0*
   8.3 Fault tolerance: *L0*
9. Quality of Service
   9.1 QoS Metrics: throughput, end-to-end delay
   9.2 QoS Level: N/A
   9.3 Cost: N/A

9.4 Quality Level: N/A

Abstract Component: *DeluxeTransactionServer*
…
Abstract Component: *EconomicTransactionServer*
…
Abstract Component: *TransactionServerManager*
…
Abstract Component: *CashierTerminal*
…
Abstract Component: *ATM*
…
Abstract Component: *CashierValidationServer*
…
Abstract Component: *CustomerValidationServer*
…

3.3 QoS-related Models
3.3.1 Critical Use Case Model (Function/Interface Level)

- Disjunctive Normal Form
  CriticalUseCaseModel: one-of (CriticalUseCaseModel1, CriticalUseCaseModel2, CriticalUseCaseModel3)

  CriticalUseCaseModel1: all (DepositMoneyCase1_1, WithdrawMoneyCase1_1, TransferMoneyCase1_1)
  CriticalUseCaseModel2: all (DepositMoneyCase1_2, WithdrawMoneyCase1_2, TransferMoneyCase1_2)
  CriticalUseCaseModel3: all (DepositMoneyCase2, WithdrawMoneyCase2, TransferMoneyCase2)

  DepositMoneyCase1_1: path_f(CashierTerminal.deposit[cp2s], DeluxeTransactionServer.deposit[cp2s], AccountDatabase.getAccount[cp2s], AccountDatabase.saveAccount[cp2s])
  DepositMoneyCase1_2: path_f (CashierTerminal.deposit[cp2s], DeluxeTransactionServer.deposit[cp2s], AccountDatabase.getAccount[cp2a], AccountDatabase.saveAccount[cp2a])
  DepositMoneyCase2: path_f (CashierTerminal.deposit[cp2s], EconomicTransactionServer.deposit[cp2s])
  WithdrawMoneyCase1_1: path_f (CashierTerminal.withdraw[cp2s], DeluxeTransactionServer.withdraw[cp2s], AccountDatabase.getAccount[cp2s], AccountDatabase.saveAccount[cp2s])
  WithdrawMoneyCase1_2: path_f (CashierTerminal.withdraw[cp2s], DeluxeTransactionServer.withdraw[cp2s], AccountDatabase.getAccount[cp2a], AccountDatabase.saveAccount[cp2a])
  WithdrawMoneyCase2: path_f (CashierTerminal.transfer[cp2s], EconomicTransactionServer.transfer[cp2s])
  TransferMoneyCase1_1: path_f (CashierTerminal.transfer[cp2s], DeluxeTransactionServer.transfer[cp2s], AccountDatabase.getAccount[cp2s], AccountDatabase.saveAccount[cp2s])
  TransferMoneyCase1_2: path_f (CashierTerminal.transfer[cp2s], DeluxeTransactionServer.transfer[cp2s], AccountDatabase.getAccount[cp2a], AccountDatabase.saveAccount[cp2a])
  TransferMoneyCase2: path_f (CashierTerminal.transfer[cp2s], EconomicTransactionServer.transfer[cp2s])

- Constraint Expression
    - Default Constraint
      default (CriticalUseCase: CriticalUseCase3)

3.3.2 Architecture Model and Critical Use Case Model Mapping (Function/Interface Level)
map (BankCase1_1: CriticalUseCaseModel3)
map (BankCase2_1: CriticalUseCaseModel1)
map (BankCase2_2: CriticalUseCaseModel2)
map (BankCase3_1: CriticalUseCaseModel3)
map (BankCase4_1: CriticalUseCaseModel1)
map (BankCase4_2: CriticalUseCaseModel2)

3.3.3 QoS Composition and Decomposition Model (QCDM)
QCDM: one-of(CriticalUseCaseModel1, CriticalUseCaseModel2, CriticalUseCaseModel3)

- CriticalUseCaseModel1

1) QoS Composition Model
1.1) QoS Composition Rules for throughput
System_throughput = CriticalUseCaseModel1_throughput
CriticalUseCaseModel1_throughput = min (DepositMoneyCase1_1_throughput, WithdrawMoneyCase1_1_throughput, TransferMoneyCase1_1_throughput)
1/DepositMoneyCase1_1_throughput = 1/CashierTerminal.deposit_throughput + 1/DeluxeTransactionServer.deposit_throughput + 1/AccountDatabase.getAccount_throughput + 1/AccountDatabase.saveAccount_throughput
1/WithdrawMoneyCase1_1_throughput = 1/CashierTerminal.withdraw_throughput + 1/DeluxeTransactionServer.withdraw_throughput + 1/AccountDatabase.getAccount_throughput + 1/AccountDatabase.saveAccount_throughput
1/TransferMoneyCase1_1_throughput = 1/CashierTerminal.transfer_throughput + 1/DeluxeTransactionServer.transfer_throughput + 1/AccountDatabase.getAccount_throughput + 1/AccountDatabase.saveAccount_throughput
1.2) QoS Composition Rules for endToEndDelay
SystemQoS.endToEndDelay = CriticalUseCaseModel1_endToEndDelay
CriticalUseCaseModel1_endToEndDelay = max (DepositMoneyCase1_1_endToEndDelay, WithdrawMoneyCase1_1_endToEndDelay, TransferMoneyCase1_1_endToEndDelay)
DepositMoneyCase1_1_endToEndDelay = sum(CashierTerminal.deposit_endToEndDelay, DeluxeTransactionServer.deposit_endToEndDelay, AccountDatabase.getAccount_endToEndDelay, AccountDatabase.saveAccount_endToEndDelay)
WithdrawMoneyCase1_1_endToEndDelay = sum(CashierTerminal.withdraw_endToEndDelay, DeluxeTransactionServer.withdraw_endToEndDelay, AccountDatabase.getAccount_endToEndDelay, AccountDatabase.saveAccount_endToEndDelay)
TransferMoneyCase1_1_endToEndDelay = sum(CashierTerminal.transfer_endToEndDelay, DeluxeTransactionServer.transfer_endToEndDelay, AccountDatabase.getAccount_endToEndDelay, AccountDatabase.saveAccount_endToEndDelay)
2) QoS Decomposition Model
2.1) QoS Decomposition Rules for throughput
CashierTerminal.deposit_throughput > System_throughput
CashierTerminal.withdraw_throughput > System_throughput
CashierTerminal.transfer_throughput > System_throughput

DeluxeTransactionServer.deposit_throughput > System_throughput
DeluxeTransactionServer.withdraw_throughput > System_throughput
DeluxeTransactionServer.transfer_throughput > System_throughput
AccountDatabase.getAccount_throughput > System_throughput
AccountDatabase.saveAccount_throughput > System_throughput

2.2) QoS Decomposition Rules for endToEndDelay

CashierTerminal.deposit_endToEndDelay < System_endToEndDelay
CashierTerminal.withdraw_endToEndDelay < System_endToEndDelay
CashierTerminal.transfer_endToEndDelay < System_endToEndDelay
DeluxeTransactionServer.deposit_endToEndDelay < System_endToEndDelay
DeluxeTransactionServer.withdraw_endToEndDelay < System_endToEndDelay
DeluxeTransactionServer.transfer_endToEndDelay < System_endToEndDelay
AccountDatabase.getAccount_endToEndDelay < System_endToEndDelay
AccountDatabase.saveAccount_endToEndDelay < System_endToEndDelay

- CriticalUseCaseModel2
  …
- CriticalUseCaseModel3
  …

APPENDIX K: Acronyms

ACIM: Abstract Component Interface Model
ACM: Abstract Component Model
AMDNF: Architecture Model in Disjunctive Normal Form
AMHF: Architecture Model in Hierarchical Form
AMNF: Architecture Model in Normalized Form
AMM: Architecture Model Mapping
BNF: Backus-Naur Form
CBSD: Component-based Software Development
CONT: Commercial Off-the-Net
COTS: Commercial Off-the-Shelf
CUCM: Critical Use Case Model
DCS: Distributed Computing Systems
DFIM: Design Feature Interaction Model
DSL: Domain Specific Language
IM: Interface Model
MDA: Model Driven Architecture
MM: Multiplicity Model
MMSL: System-Level Multiplicity Model
MMCL: Component-level Multiplicity Model
PDM: Platform Dependent Model
PIM: Platform Independent Model
PLP: Product Line Practice
QoS: Quality of Service
QRM: QoS Requirement Model
UA: UniFrame Approach
UCM: Use Case Model
UDSL: UniFrame Domain Specific Language
UGDM: UniFrame Generative Domain Model
UGDP: UniFrame UGDM Development Process
UQOS: UniFrame QoS Framework
URDS: UniFrame Resource Discovery Service
USGI: UniFrame System Generation Infrastructure
USGPF: UniFrame System-Level Generative Programming Framework

LIST OF REFERENCES

LIST OF REFERENCES

[AAG01] J. Ø. Aagedal. Quality of Service Support in Development of Distributed Systems. PhD thesis, Department of Informatics, Faculty of Mathematics and Natural Sciences, University of Oslo, 2001.

[APA03] Apache. Xerces Java Parser Readme. http://xml.apache.org/xerces-j/, 2003.

[APA03a] Apache Jakarta Project. Apache Tomcast. http://jakarta.apache.org/tomcat/, 2003.

[AUG95] M. Auguston. Program Behavior Model Based on Event Grammar and its Application for Debugging Automation. In *Proceedings of the 2nd Inernational Workshop on Automated and Algorithmic Debugging*, pages 277-291, 1995.

[AUG97] M. Auguston, A. Gates, M. Lujan. Defining a Program Behavior Model for Dynamic Analyzers. In *Proceedings of the 9th International Conference on Software Engineering and Knowledge Engineering, SEKE'97*, pages 257-262, 1997.

[BAT92] D. Batory, S. O'Malley. The Design and Implementation of Hierarchical Software Systems with Reusable Components. *ACM Transactions on Software Engineering and Methodology*, October 1992.

[BAT95] D. Batory, L. Coglianese, M. Goodwin, S. Shafer. Creating Reference Architectures: An Example from Avionics. *Symposium on Software Reusability 1995*, Seattle, Washington.

[BAT96] D. Batory. Subjectivity and GenVoca Generators. *1996 International Conference on Software Resuse*, Orlando, Florida.

[BAT02] D. Batory, R. Lopez-Herrejon, J. Martin. Generating Product-Lines of Product-Families. *2002 Automated Software Engineering Conference*, Edinburgh, Scotland, pp 81-92.

[BAY03] Bayfront Technologies, Inc. http://www.bayfronttechnologies.com, 2003.

[BOO98] G. Booch, I. Jacobson, J. Rumbaugh, J. Rumbaugh. The Unified Modeling Language User Guide. Addison-Wesley, 1998. ISBN: 0201571684.

[BRA01] G. Brahnmath, R. Raje, A. Olson, C. Sun. Quality of Service Catalog for Software Components. Technical Report (TR-CIS-0219-01), Department of Computer and Information Science, Indiana University Purdue University Indianapolis, 2001.

[BRA02] G. Brahnmath. The UniFrame Quality of Service Framework. MS Thesis, Department of Computer & Information Science, Indiana University Purdue University Indianapolis, December 2002.

[BRA02a] G. Brahnmath, R. Raje, A. Olson, B. Bryant, M. Auguston, C. Burt. A Quality of Service Catalog for Software Components. *The Proceedings of the Southeastern Software Engineering Conference*, Huntsville, Alabama, April 2002, pages 513-520.

[BRY00] B. Bryant. Object-Oriented Natural Language Requirements Specification. In *Proceedings of ACSC 2000, the 23rd Australasian Computer Science Conference, January 30-February 4, 2000, Canberra, Australia*, January 2000, pages 24-30.

[BRY02] B. Bryant, B. Lee. Two-Level Grammar as an Object-Oriented Requirements Specification Language, *Proceedings (CR-ROM) of 35th Hawaii International Conference on System Sciences, 2002*, page 10. http://www.hicss.hawaii.edu/HICSS_35/HICSSpapers/PDFdocuments/STDSLO1.pdf.

[BRY02a] B. Bryant, C. Burt, M. Auguston, R. Raje, A. Olson. Formal Specification of Generative Component Assembly Using Two-Level Grammar. *Proceedings of SEKE 2002, Fourteenth International Conference on Software Engineering and Knowledge Engineering*, July 15-19, 2002, Sant'Angelo d'Ischia, Italy

[BUS96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. Pattern-Oriented Software Architecture. A System of Patterns. John Wiley & Sons Ltd, Chichester, UK, 1996.

[CAO02] F. Cao, B. Bryant, R. Raje, M. Auguston, A. Olson, C. Burt. Component Specification and Wrapper/Glue Code Generation with Two-Level Grammar using Domain Specific Knowledge. *Proceedings of ICFEM 2002, 4th International Conference on Formal Engineering Methods*, Shanghai, China, October 2002. Springer-Verlag Lecture Notes in Computer Science, Vol. 2495, 2002, pp. 103-107.

[CAO03] F. Cao, Z. Huang, B. Bryant, R. Raje, A. Olson, M. Auguston, C. Burt. To be appear on the *Proceedings of the 2003 International Conference on Software Engineering Research and Practice*. SERP'03: June 23-26, 2003, Las Vegas, Nevada, USA.

[CLE88] J. C. Cleaveland. Building application generators. *IEEE Software*, pages 25–33, July 1988.

[CLE01] P. Clements, P. Donohoe, K. Kang, L. Northrop. Fifth Product Line Practice Workshop Report. September, 2001.
http://www.sei.cmu.edu/publications/documents/01.reports/01tr027.html.

[COH99] S. Cohen. From Product Line Architectures to Products. *Position paper for the ECOOP'99 Workshop on Object-Technology for Product-Line Architectures*, Lisbon, Portugal, June 1999. http://www.esi.es/Projects/Reuse/Praise/pdf/ses2-1.pdf.

[COH00] S. Cohen, B. Gallagher, M. Fisher, L. Jones, R. Krut, L. Northrop, W. O'Brien, D. Smith, A. Soule. Third DoD Product Line Practice Workshop Report. July 2000. http://www.sei.cmu.edu/publications/documents/00.reports/00tr024.html.

[CZA99] K. Czarnecki, U.W. Eisenecker. Components and Generative Programming. *Proceedings of the Joint European Software Engineering Conference and ACM SIGSOFT International Symposium on the Foundations of Software Engineering* (ESEC/FSE 99, Toulouse, Frankreich, September 1999). Springer-Verlag, 1999.http://www-ia.tu-ilmenau.de/~czarn/esec99/esec99.pdf.

[CZA99a] K. Czarnecki. DEMRAL: Domain Engineering Method for Developing Reusable Algorithmic. http://www-ia.tu-ilmenau.de/~czarn/. 1999.

[CZA00] K. Czarnecki, U.W. Eisenecker. Generative Programming: Methods, Tools, and Applications. Addison-Wesley, 2000.

[GAM95] E. Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns Elements of Reusable Object-Orientated Software. Addison-Wesley, 1995.

[GME] Generic Modeling Environment. Institute for Software Integrated Systems. Vanderbilt University. http://www.isis.vanderbilt.edu/Projects/gme/default.html.

[HUA02] Z. Huang, R. Raje, A. Olson, B. Bryant, M. Auguston, C. Burt, C. Sun. Unified Approach for System-Level Generative Programming. *Proceedings of the IEEE Fifth International Conference on Algorithms and Architectures for Parallel Processing*, Beijing, China, October 2002, pp. 136-142.

[IBM02] IBM. IBM WebSphere V4.0 Advanced Edition Handbook. Chapter 17, March 2002. http://www.redbooks.ibm.com/redpieces/pdfs/sg246176.pdf.

[ISO86] Quality Vocabulary. International Organization for Standardization, Geneva. ISO 8402: 1986, page 8.

[KAN90] K. Kang, S. Cohen, J. Hess, W. Nowak, S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report, CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, November 1990.

[LEE02] B. Lee, B. Bryant. Automated Conversion from Requirements Documentation to an Object-Oriented Formal Specification Language. *Proceedings of SAC 2002, the 2002 ACM Symposium on Applied Computing, March 11-14, 2002, Madrid, Spain, 2002,* pp. 932-936.

[LEE02a] B. Lee, B. Bryant. Automation of Software System Development Using Natural Language Processing and Two-Level Grammar. *Proceedings of the 2002 Monterey Workshop on Radical Innovations Software and Systems Engineering in the Future*, Venice, Italy, October 2002.

[LUQ01] Luqi, V. Berzins, J. Ge, M. Shing, M. Auguston, B. Bryant, B. Kin. DCAPS - Architecture for Distributed Computer Aided Prototyping System. In *Proceedings of the 12th IEEE International Workshop on Rapid System prototyping, pp.103-109, June 25-27, 2001, Monterey Beach Resort, California, USA*, IEEE Computer Society Press, 2001.

[MAY02] S. Mayo. Web Services: How Will Professional Services Firms Compete for This Multibillion-Dollar Opportunity. IDC, March 2002.

[MS98] Microsoft Corporation. DCOM Specifications.
URL: -http://www.microsoft.com/oledev/olecom, 1998.

[NEI80] J. Neighbors. Software Construction Using Components. PhD Thesis. University of California at Irvine, 1980, UCI ICS technical report TR-160.

[NEI03] J. Neighbors. Draco 1.2 Users Manual.
http://www.bayfronttechnologies.com/manual.htm, 2003.

[NET03] .NET, Microsoft Corporation. http://www.microsoft.com/net/, 2003.

[OMG99] Object Management Group. CORBA Components. Technical report, Object Management Group TC Document orbos/99-02-05, March 1999. http://www.omg.org/cgi-bin/doc?orbos/99-02-05.

[OMG01] Object Management Group (OMG). Model Driven Architecture: A Technical Perspective. Technical Report, OMG Document No. ab/2001-02-01/04, February 2001. ftp://ftp.omg.org/pub/docs/ab/01-02-04.pdf.

[OMG03] Object Management Group (OMG). UML Notation Guide. formal/03-03-10 (UMl 1.5 chapter 3 - UML Notation Guide). http://www.omg.org/cgi-bin/doc?formal/03-03-10, 2003.

[ORA03] Oracle. http://www.oracle.com, 2003.

[ORF98] R. Orfali, D. Harkey. Client/Server Programming with JAVA and CORBA. The second edition. John Wiley & Sons, Inc., 1998.

[RAJ00] R. Raje. UMM: Unified Meta-object Model. *Proceedings of 4th IEEE International Conference on Algorithms and Architecture for Parallel Processing, ICA3PP'2000, pp: 454-465,* Hong Kong, 2000.

[RAJ01] R. Raje, B. Bryant, M. Auguston, A. Olson, C. Burt. A Unified Approach for the Integration of Distributed Heterogeneous Software Components. *Proceedings of the 2001 Monterey Workshop on Engineering Automation for Software Intensive System Integration, Monterey, California, 2001,* pp: 109-119.

[RAJ02] R. Raje, B. Bryant, A. Olson, M. Auguston, C. Burt. A quality-of-service-based framework for creating distributed heterogeneous software components. *Concurrency and Computation: Practice and Experience*, Volume 14, Issue 12, 2002. Pages: 1009-1034.

[SEI96] J. Seigel. CORBA Fundamentals and Programming. John Wiley & Sons, Inc., 1996.

[SEI02] Software Engineering Institute, Carnegie Mellon University. The Product Line Approach Initiative. http://www.sei.cmu.edu/plp/plp_init.html, 2002.

[SEI02a] Software Engineering Institute, Carnegie Mellon University. A Framework for Software Product Line Practice-Version 3.0.
http://www.sei.cmu.edu/plp/framework.html, 2002.

[SEI02b] Software Engineering Institute, Carnegie Mellon Univeristy. Organization Domain Modeling. http://www.sei.cmu.edu/str/descriptions/odm.html, 2002.

[SEI03] Software Engineering Institute, Carnegie Mellon University. Feature-Oriented Domain Analysis. http://www.sei.cmu.edu/domain-engineering/FODA.html, 2003.

[SHA96] M. Shaw, D. Garlan. Software Architecture: Perspectives on a Emerging Discipline. Prentice Hall, Englewood Cliffs, NJ, 1996. ISBN: 0-13-182957-2

[SIM96] M. Simos, et al. *Software Technology for Adaptable Reliable Systems (STARS) Organization Domain Modeling (ODM) Guidebook* Version 2.0 (STARS-VC-A025/001/00). Manassas, VA: Lockheed Martin Tactical Defense Systems, 1996.

[SIR02] N. Siram. An Architecture for the UniFrame Resource Discovery Service. MS Thesis. Indiana University Purdue University Indianapolis, March 2002.

[STE00] B**.** Stearns. JavaBeans 101, Part I. October 2000.
http://developer.java.sun.com/developer/onlineTraining/Beans/bean01/index.html**.**

[SUN02] C. Sun, R. Raje, A. Olson, M. Auguston, B. Bryant, C. Burt, Z. Huang. Composition and Decomposition of Quality of Service Parameters in Distributed Component-based Systems. To appear in *Prodeedings of the Fifth International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP 2002)*.

[SUN03] C. Sun, R. Raje, A. Olson, B. Bryant, C. Burt and M. Auguston. A Composition Model for the Response Time and Throughput in Distributed Component-Based Systems. Technical Report, TR-0808-03, Department of Computer and Information Science, IUPUI. May 8, 2003.

[SM01] Sun Microsystems. Java^{TM} 2 Platform Enterprise Edition Specification, Version 1.3. Sun Microsystems, August 2001. http://java.sun.com/j2ee/j2ee-1_3-fr-spec.pdf

[SM02] Sun Microsystems. The J2EE Turorial^{TM}. Sun Microsystems, April 24, 2002. http://java.sun.com/j2ee/tutorial/.

[SM02a] Sun Microsystems**.** Developing Enterprise Applications Using the J2EE^{TM} Platform. Sun Microsystems, August 2002.
http://developer.java.sun.com/developer/onlineTraining/J2EE/Intro2/j2ee.html

[SM03] Sun Microsystems. Java 2 Platform, Standard Edition (J2SE), v1.4.0. http://java.sun.com/j2se/1.4/, 2003.

[SM03a] Sun Microsystems. Javabeans Component Architecture Documentation. http://java.sun.com/products/javabeans/docs/, 2003.

[SM03b] Sun Microsystems. JDBC^{TM} API.
http://java.sun.com/products/jdk/1.2/docs/guide/jdbc/, 2003.

[SM03c] Sun Microsystems. Java Transaction API (JTA).
http://java.sun.com/products/jta/, 2003.

[SM03d] Sun Microsystems. The JNDI Tutorial.
http://java.sun.com/products/jndi/tutorial/, 2003.

[SM03e] Sun Microsystems. J2EE Connector Architecture.
http://java.sun.com/j2ee/connector/, 2003.

[SM03f] Sun Microsystems**.** Java API for XML Processing (JAXP) Documentation.
http://java.sun.com/xml/jaxp/docs.html, 2003.

[SM03g] Sun Microsystems**.** Java Remote Method Invocation (RMI).
http://java.sun.com/products/jdk/rmi/

[SZY99] C. Szyperski. Component Software: Beyond Object-Oriented Programming. Addison-Wesley, ISBN 0-201-17888-5, 1999, page 34.

[VAN00] A. van Deursen, P. Klint, and J. Visser. Domain-specific languages: An annotated bibliography. ACM SIGPLAN Notices, 35(6):26--36, June 2000.

[VAN02] A. van Deursen, P. Klint. Domain-Specific Language Design Requires Feature Descriptions. *Journal of Computing and Information Technology*, 10(1):1-17, 2002.

[VAR02] C. Varghese. Examining, Documenting, and Modeling the Problem Space of a Variable Domain. MS Thesis. Indiana University Purdue University Indianapolis, June 2002.

[WEB02] The Web Services Community Portal, http://www.webServices.org, 2002.

[WWW03] World Wide Web. HTTP - Hypertext Transfer Protocol. http://www.w3.org/Protocols/, 2003.

[YOU95] D. Young. Object-Orientated Programming with C++ and OSF/Motif. Prentice-Hall, 1995.

[ZHA02] W. Zhao. Two-Level Grammar as the Formalism for Middleware Generation in Internet Component Broker Organizations. *Proc. of GPCE'2002 Young Research Workshop.* Pittsburgh, PA, October 2002. http://www.cs.uni-essen.de/dawis/conferences/GCSE_SAIG_YRW2002/submissions/final/Zhao.pdf